Chapter 15

Some Modern Architectectures

RISC versus CISC

- RISC designed for speed
- CISC designed for power
- CISC attempts to minimize the *semantic gap.*
- CISC has complex instructions such as a block copy which executes a copy loop in microcode.
- RISC performs a block copy by executing a loop at the machine level.

Compilers often do not take advantage of complex instructions available on a CISC. The complex instruction often does match the semantics of the high-level construct. So the compiler does not use it.

FIGURE 15.1

a) C++ code

```
for (int x = 0; x < 10; x++)
  cout << "hello\n";
```

b) Efficient assembler code (optimal instruction set)

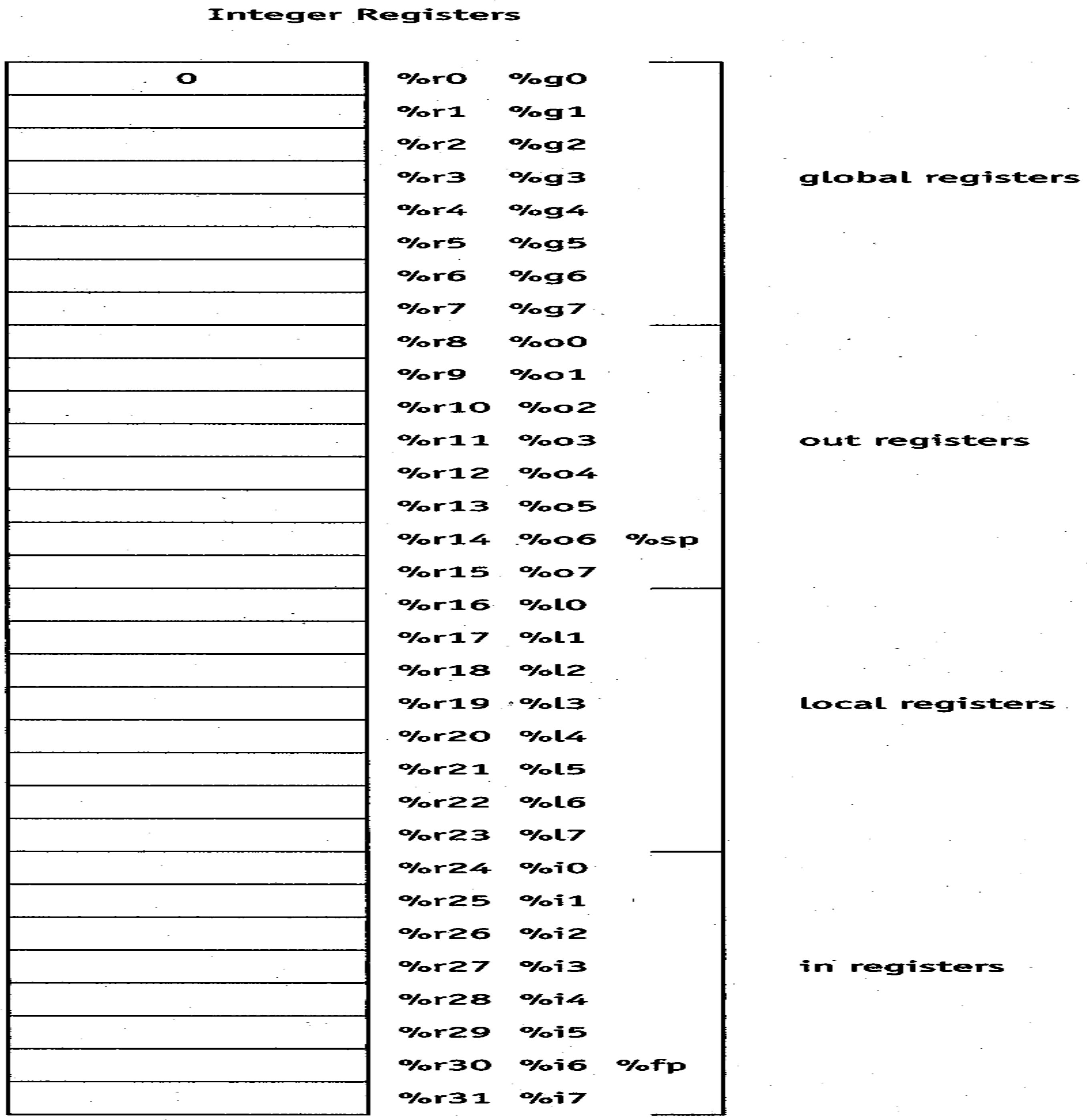
```
     ldc  10
     sect           ; load 10 into ct register
  @L0:
        .
        .           ; body of loop
        .
     dect           ; decrement ct, skip next inst if ct == 0
     ja @L0
```

c) Inefficient assembler code (optimal instruction set)

```
       ldc   0      ; allocate and initialize x
       push         ; assume relative address of x is -1

@L0:   ldr   -1     ; exit test
       push
       ldc    10
       scmp         ; compare x and 10
       jzop @L1     ; jump if x >= 10
         .
         .          ; loop body
         .

       ldc    1     ; increment x
       addr -1
       str  -1

       ja     @L0
@L1:   dloc 1       ; deallocate x
```

The SPARC is a RISC architecture.

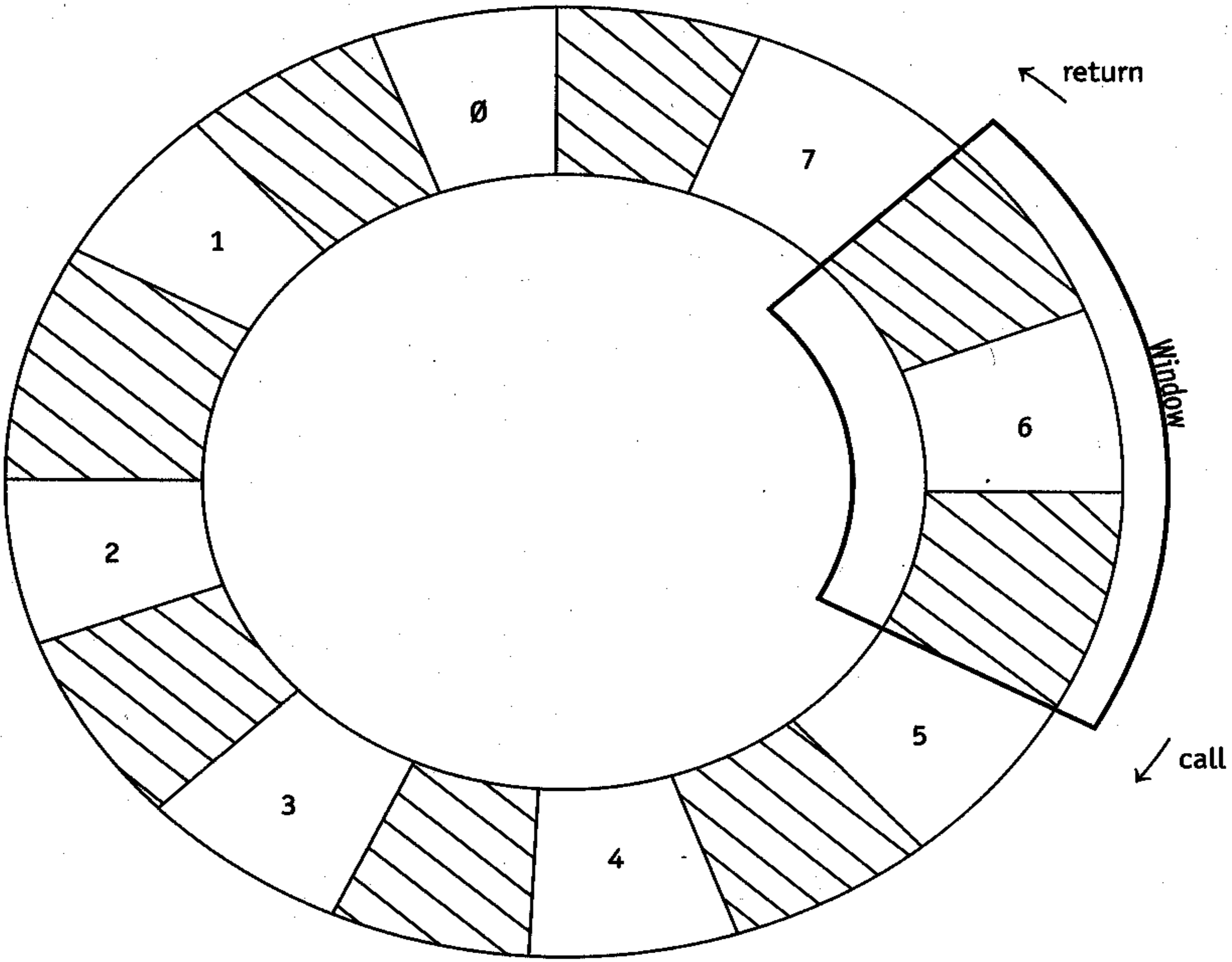
FIGURE 15.2


Integer Registers
0
%r0 %g0
%r1 %g1
%r2 %g2
%r3 %g3
%r4 %g4
%r5 %g5
%r6 %g6
%r7 %g7
global registers
%r8 %o0
%r9 %o1
%r10 %o2
%r11 %o3
%r12 %o4
%r13 %o5
%r14 %o6 %sp
%r15 %o7
out registers
%r16 %l0
%r17 %l1
%r18 %l2
%r19 %l3
%r20 %l4
%r21 %l5
%r22 %l6
%r23 %l7
local registers
%r24 %i0
%r25 %i1
%r26 %i2
%r27 %i3
%r28 %i4
%r29 %i5
%r30 %i6 %fp
%r31 %i7
in registers

The SPARC uses multiple sets of registers to speed up function calls and returns. Registers don’t have to be saved and restored on function call and return.

FIGURE 15.3

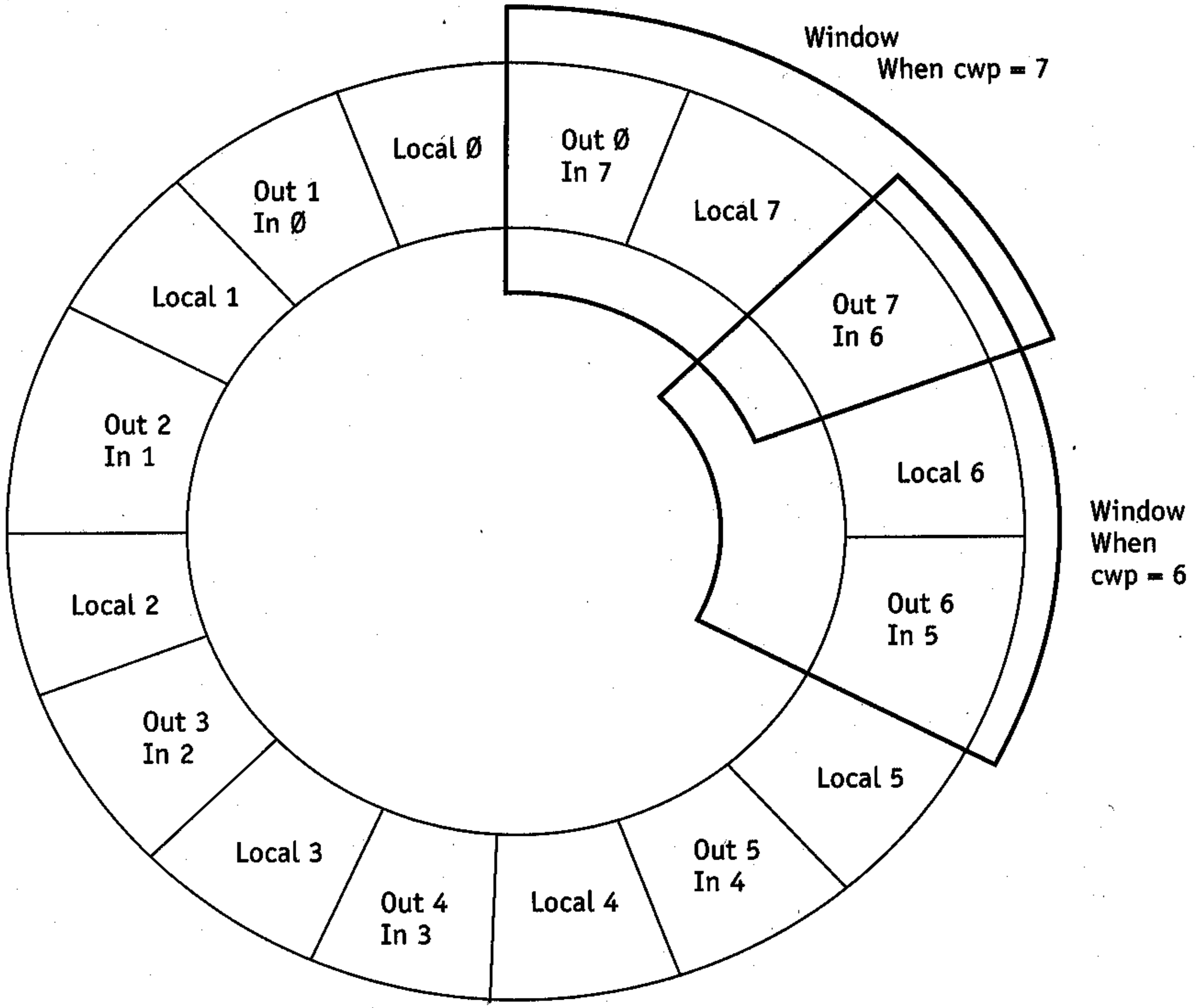
REGISTER WINDOW



Overlapping register sets make for efficient parameter passing.

The output registers of the calling function become the input registers of the called function.

FIGURE 15.4 OVERLAPPING REGISTER WINDOWS



Window overflow occurs when no register sets are available for a function call. A register set has to be *spilled* (written to main memory) to make it available for the called function.

FIGURE 15.5

WINDOW OVERFLOW

a)

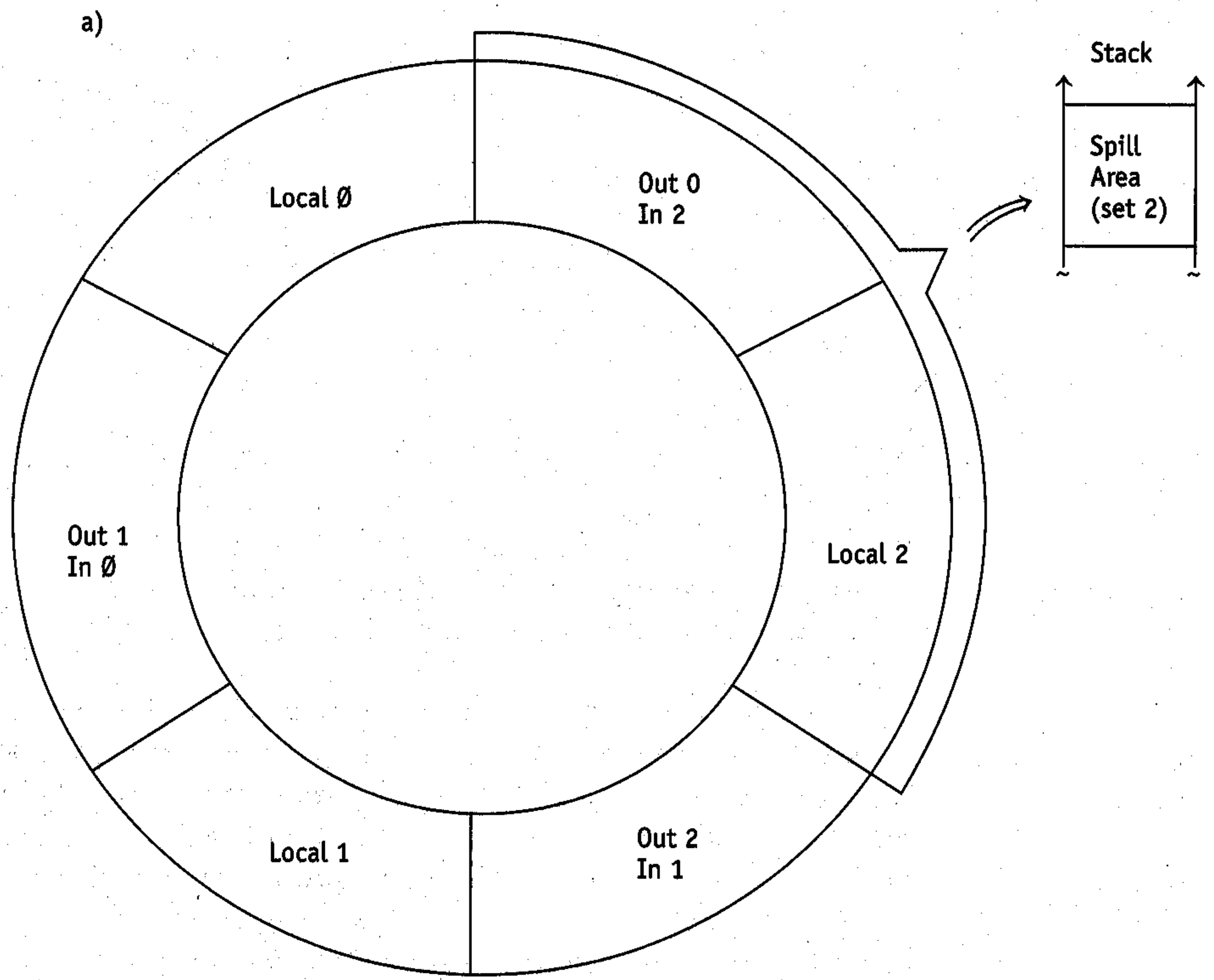
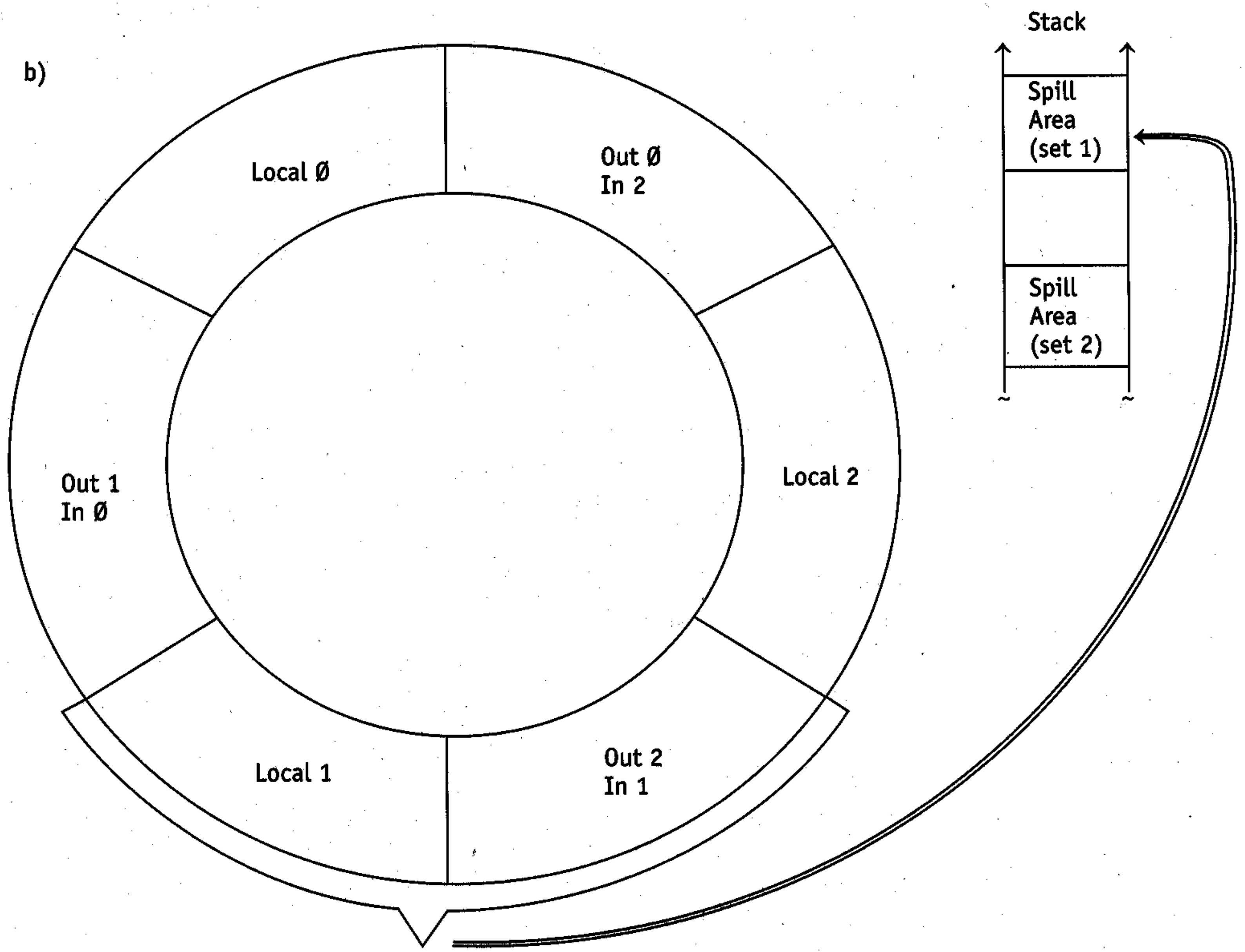


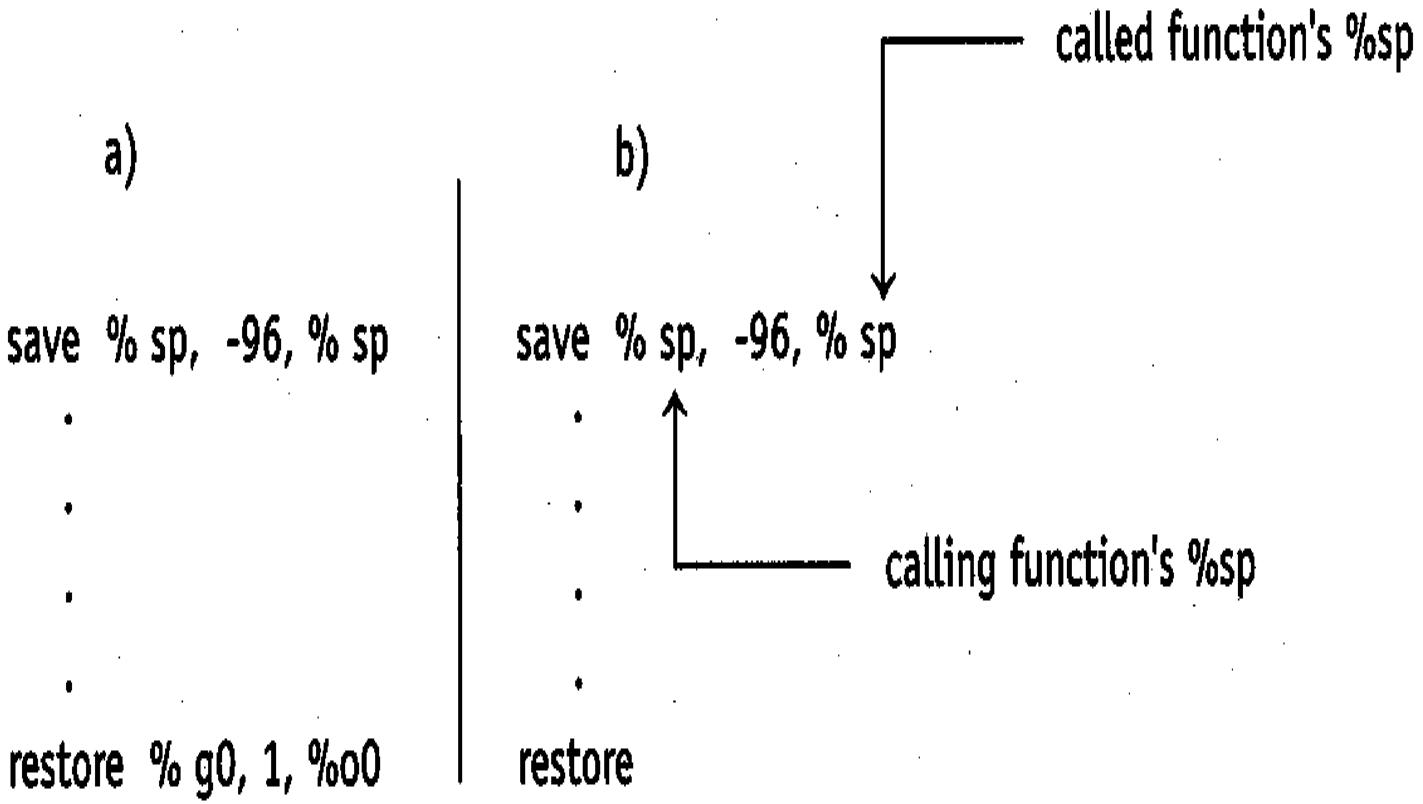
FIGURE 15.5

WINDOW OVERFLOW



The save and restore instructions trigger a register switch.

FIGURE 15.6

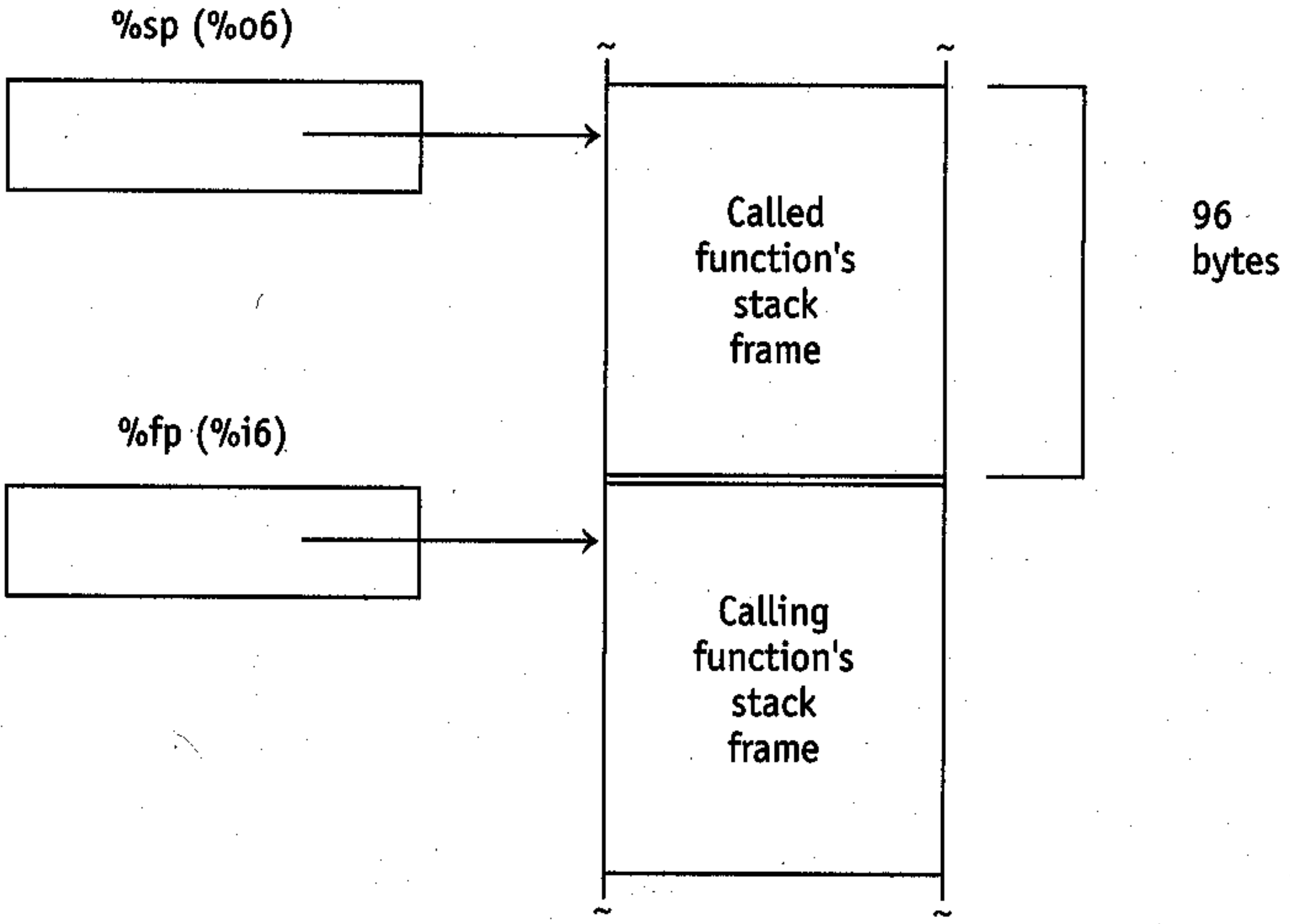


```
save  % sp,  -96, % sp
    .
    .
    .
    .
restore  % g0, 1, %o0
```

```
save  % sp,  -96, % sp
    .
    .
    .
    .
restore
```

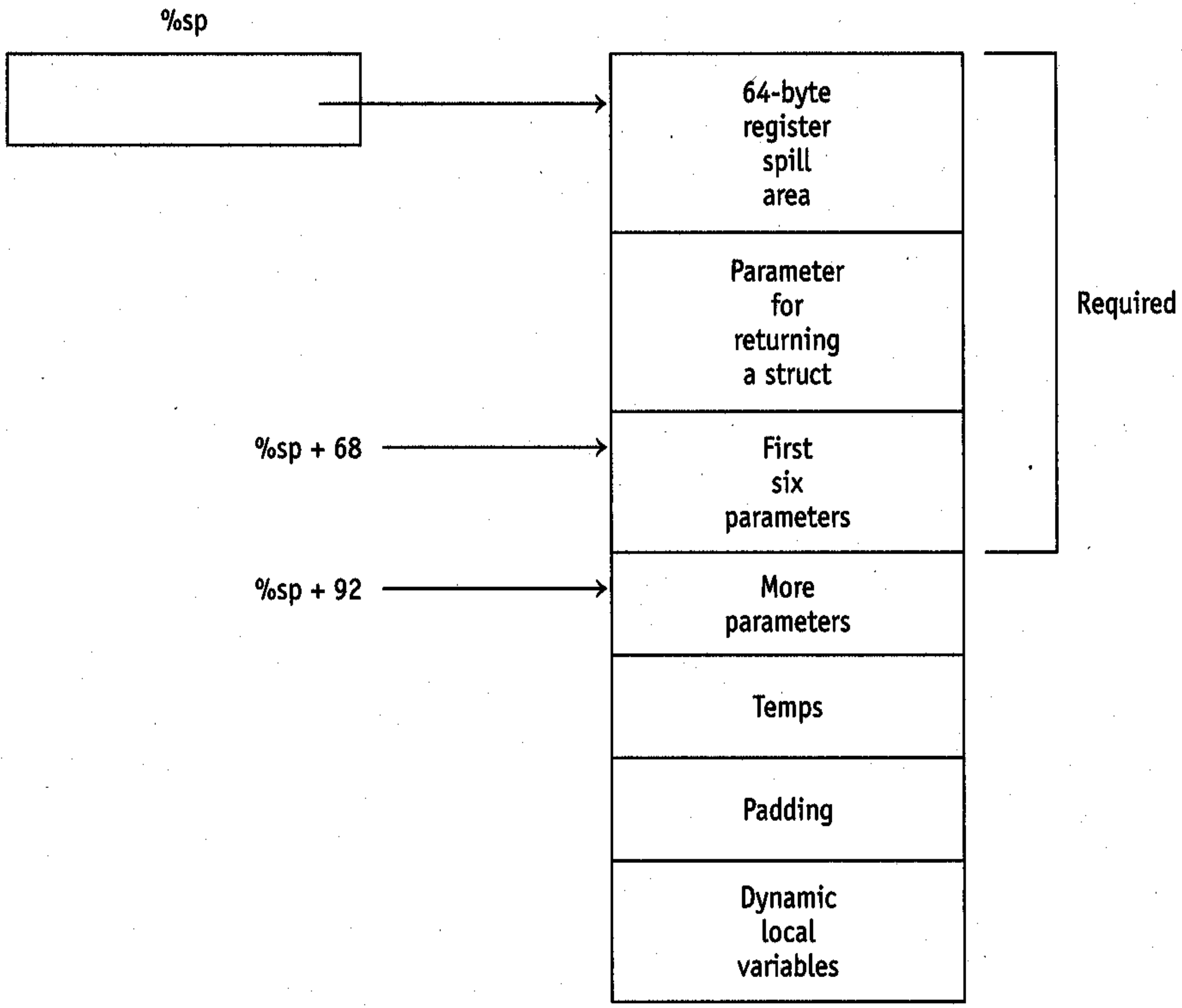
During the execution of a function, %sp points to its stack frame, %fp points to the calling function’s stack frame.

FIGURE 15.7


%sp (%o6)
%fp (%i6)
Called function's stack frame
Calling function's stack frame
96 bytes

If an item’s address is passed, the item must be in memory (so that it has an address). This is why a stack frame has an area for parameters, although they are usually passed in registers.

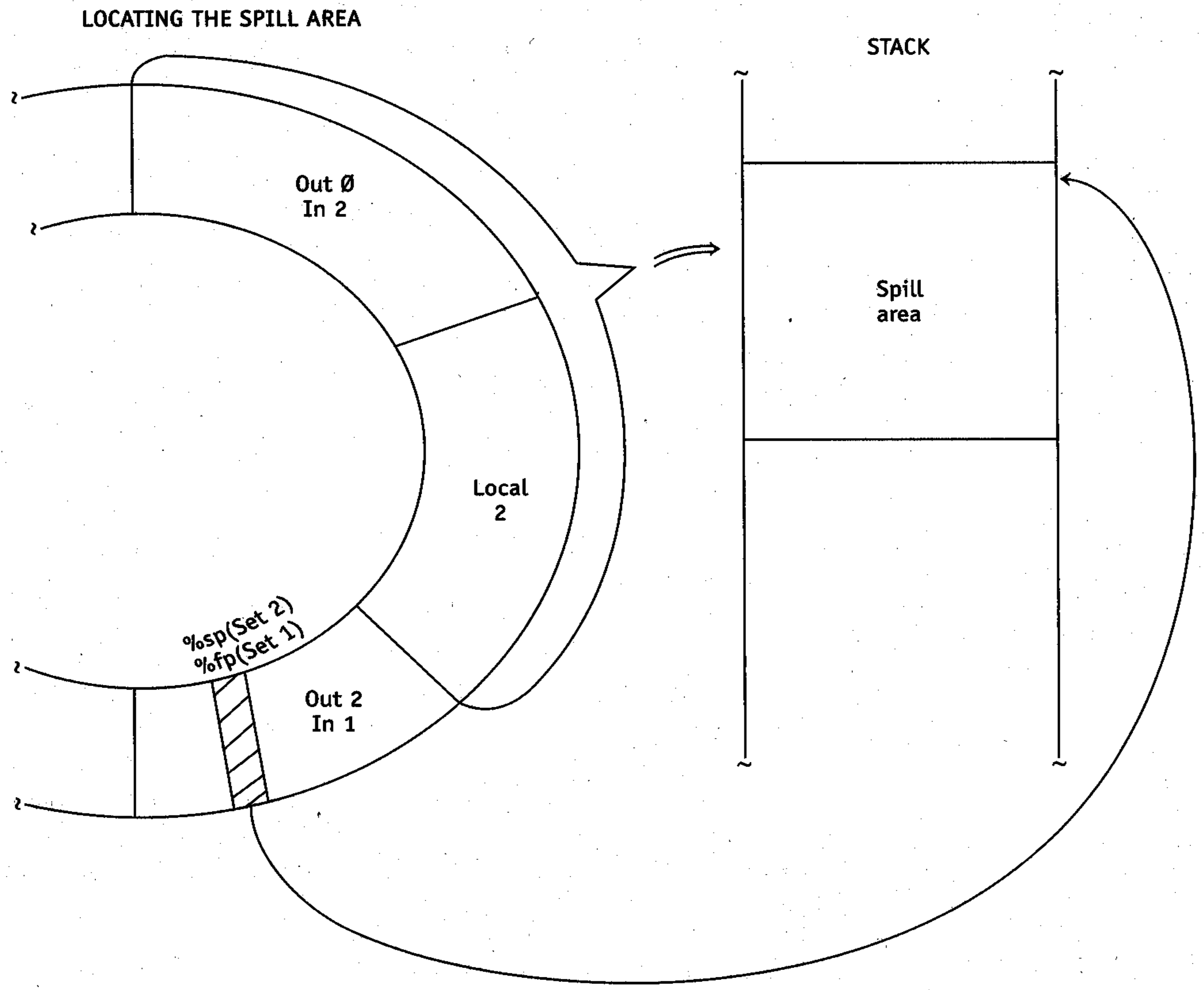
FIGURE 15.8


%sp
64-byte register spill area
Parameter for returning a struct
%sp + 68
First six parameters
Required
%sp + 92
More parameters
Temps
Padding
Dynamic local variables

The spill area for a function is in its stack frame.

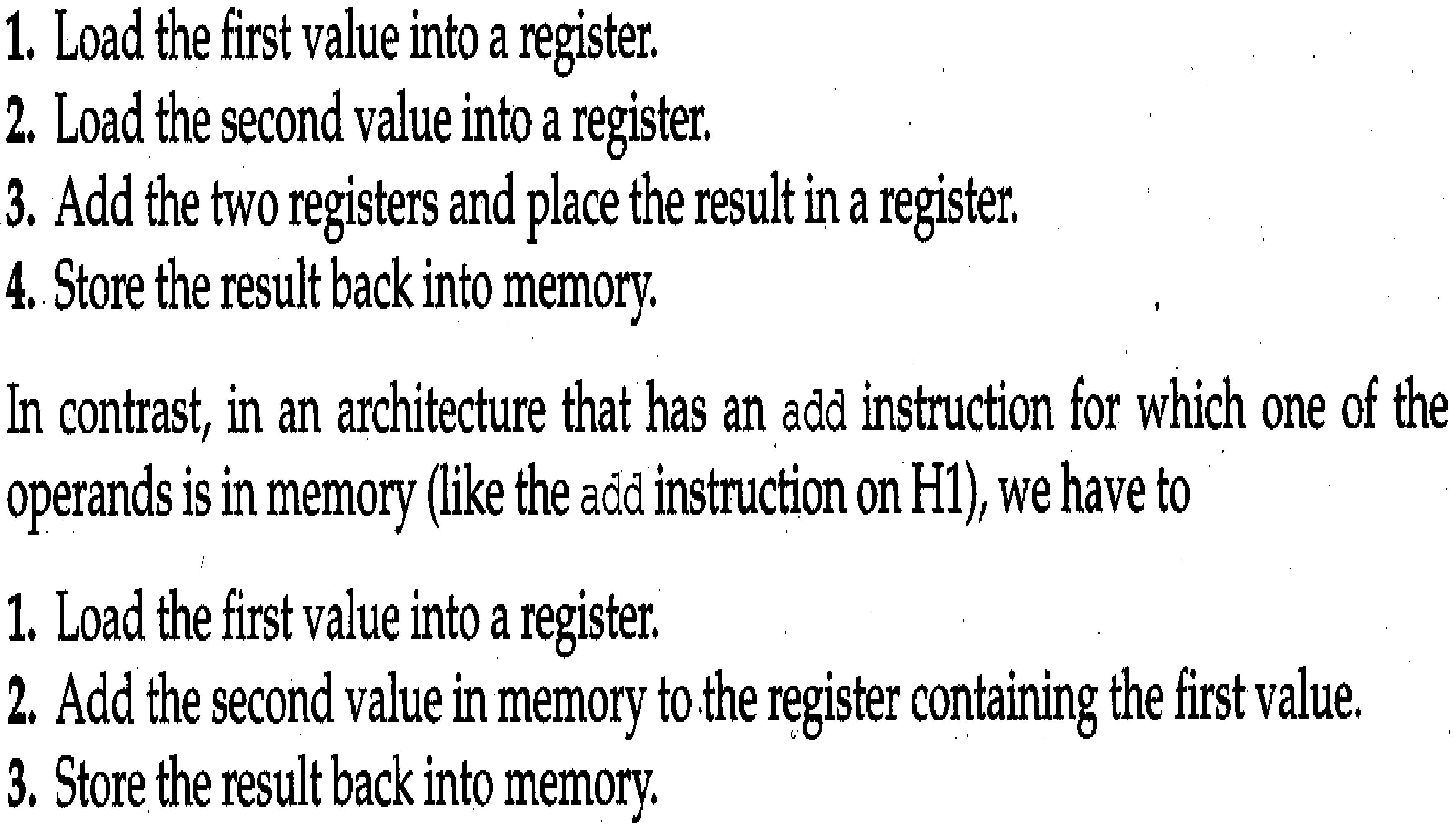
A *window underflow* occurs on return to a function whose register set was spilled (which now has to be restored from the spill area).

FIGURE 15.9


LOCATING THE SPILL AREA
STACK
Out Ø
In 2
Spill
area
Local
2
%sp(Set 2)
%fp(Set 1)
Out 2
In 1

The SPARC has a *load/store architecture*. That is, only the load and store instructions access main memory.

Code for load/store architecture



1. Load the first value into a register.
2. Load the second value into a register.
3. Add the two registers and place the result in a register.
4. Store the result back into memory.

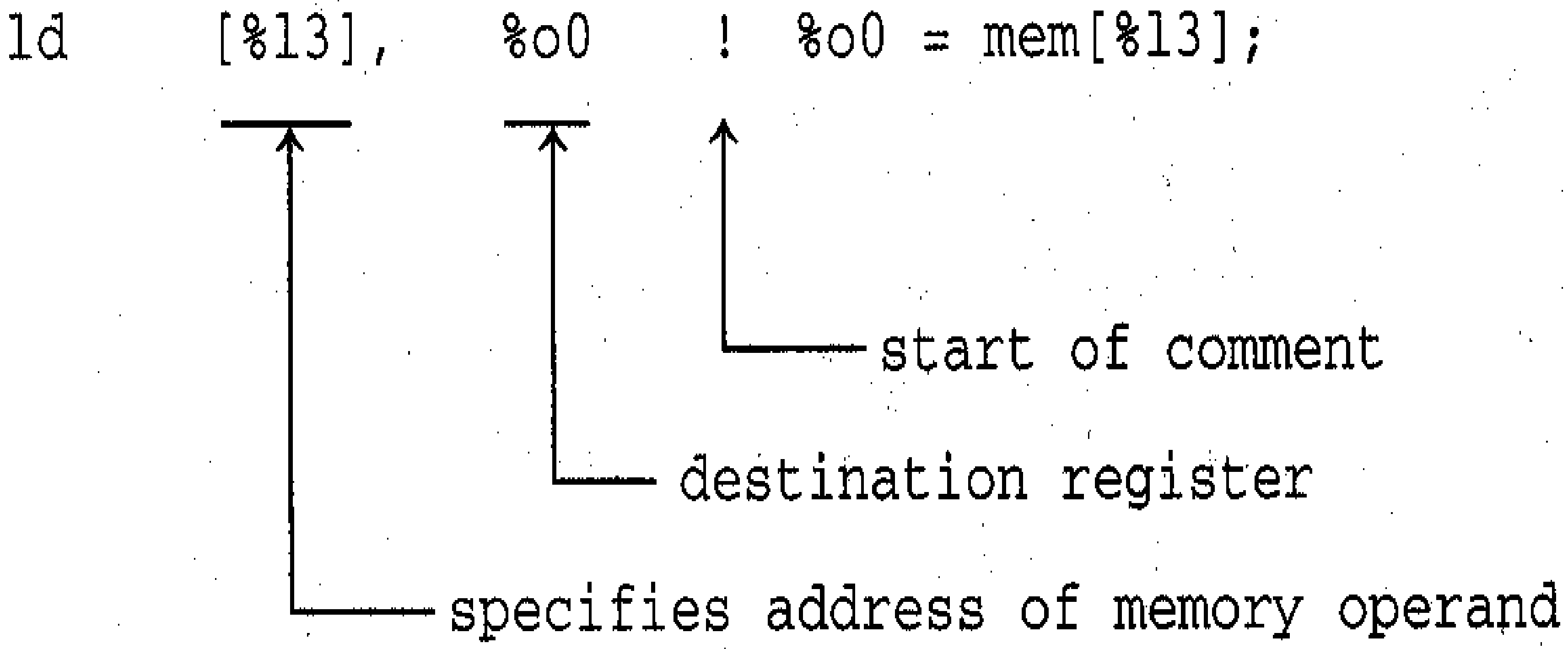
In contrast, in an architecture that has an `add` instruction for which one of the operands is in memory (like the `add` instruction on H1), we have to

1. Load the first value into a register.
2. Add the second value in memory to the register containing the first value.
3. Store the result back into memory.

Advantages of load/store architecture

- Makes for a smaller instruction set (we do not need all the arithmetic instructions that access main memory).
- Minimizes contention for main memory.
- More uniform instruction set which respect to execution time (this is good for pipelining).

ld instruction


ld [%l3], %o0 ! %o0 = mem[%l3];
start of comment
destination register
specifies address of memory operand

Three variations of ld

```
ld  [%sp], %o0
ld  [%sp + %l3], %o0
ld  [%sp + 68], %o0
```

FIGURE 15.10 a) Two source registers

31 30	29 25	24 19	18 14	13	12 5	4 0
op	rd	op3	rs1	0	---	rs2

b) One source register plus a displacement

31 30	29 25	24 19	18 14	13	12 0
op	rd	op3	rs1	1	simm13

op:	opcode, 11 for both ld and st
rd:	destination register
op3:	opcode, 00000 for ld, 00100 for st
rs1:	source register 1
rs2:	source register 2
simm13:	13-bit signed immediate value (holds displacement for load and store instructions)

A load must be from a word boundary (from an address divisible by 4)

```
x = *(int *)p;
```

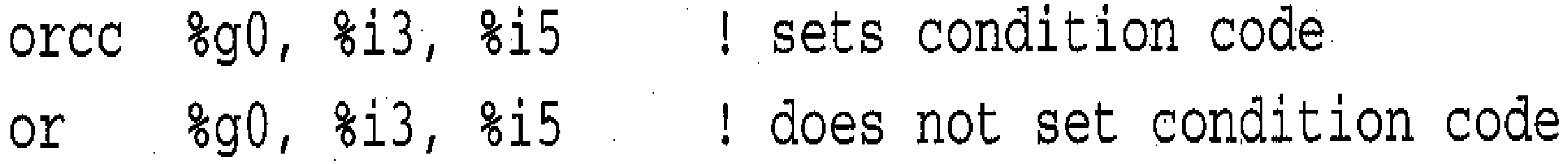
This instruction first casts the pointer to an `int` pointer and then dereferences it to access the word to which it points. However, this statement will work only if the integer is on a word boundary. If it is not, a Sun Workstation responds with the cryptic error message "Bus Error." This error message occurred when `lin` (the H1 linker) was ported from a Pentium machine to the Sun SPARC Workstation. The Pentium does not require any boundary alignments for its load instructions. Thus, it does not object to the previous statement (which appeared in the C++ code for `lin`). To fix this problem, the C++ statement was replaced with

```
memcpy(x, p, sizeof(int));
```

st instruction

```
st  %o0, [%sp]             ! mem[%sp]       = %o0;
st  %o0, [%sp + %l3]       ! mem[%sp + %l3] = %o0;
st  %o0, [%sp + 68]        ! mem[%sp + 68]  = %o0;
```

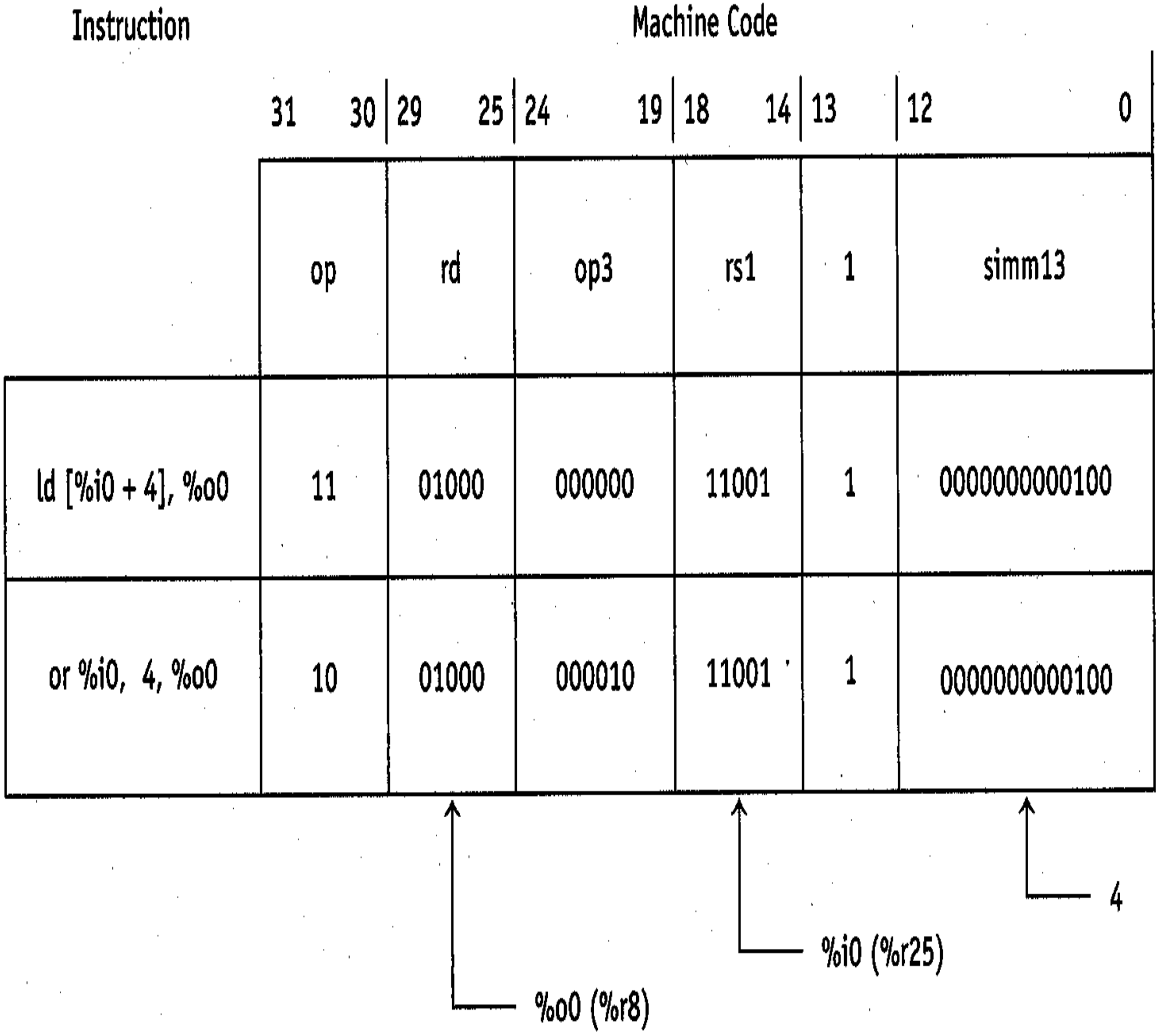
or instruction



```
orcc  %g0, %i3, %i5      ! sets condition code
or    %g0, %i3, %i5      ! does not set condition code
```

ld and **or** have essentially the same format

FIGURE 15.11



Instruction	Machine Code					
	31 30	29 25	24 19	18 14	13	12 0
	op	rd	op3	rs1	1	simm13
ld [%i0 + 4], %o0	11	01000	000000	11001	1	0000000000100
or %i0, 4, %o0	10	01000	000010	11001	1	0000000000100

To load from a symbolic label, must first load the address into a register, then use a ld instruction.

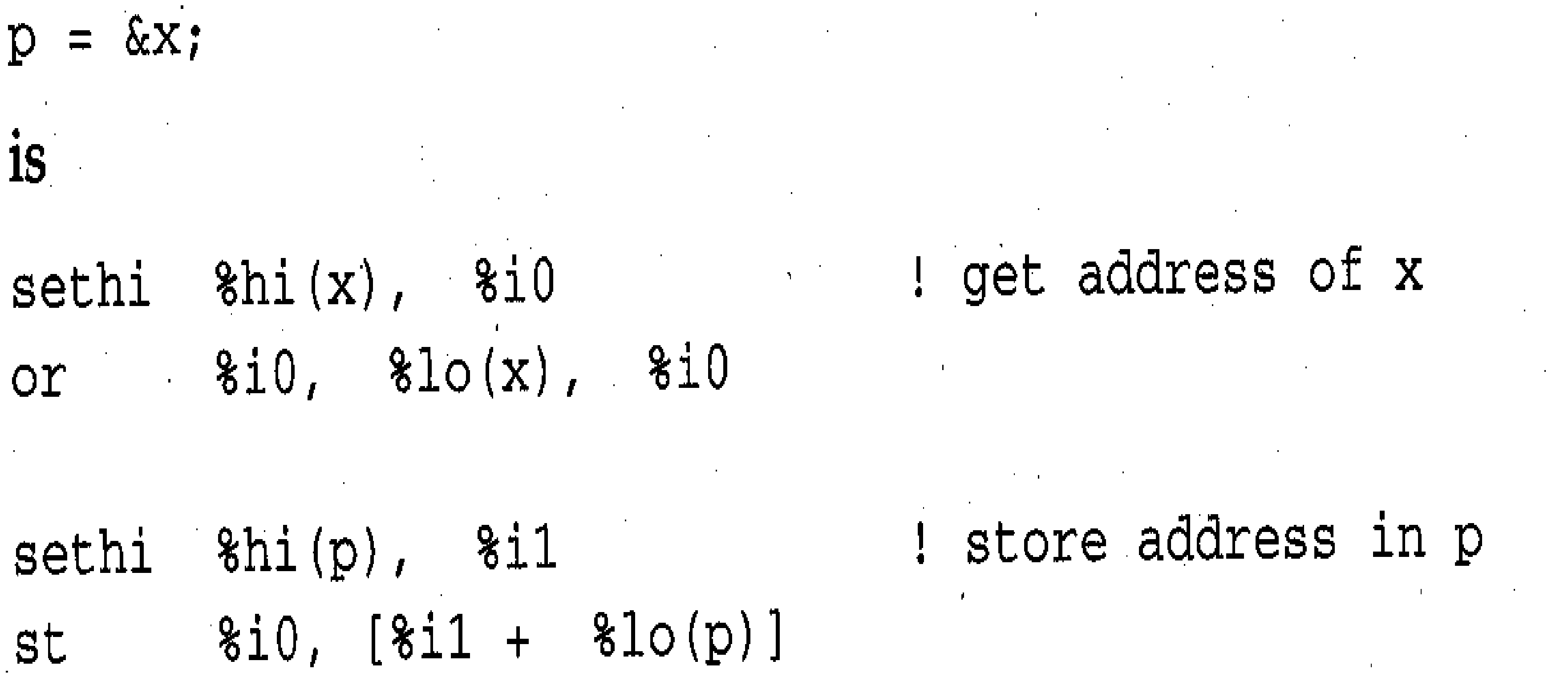
To load the address of a label into a register, use a sethi-or sequence.

FIGURE 15.12

```
        .section ".text"                    ! text section
            .
            .
            .
        sethi  %hi(x),  %i0                 ! load high 22 bits into %i0
        or     %i0,  %lo(x),  %i0           ! or 10 low bits into %i0
            .
            .
            .
        .section ".data"                    ! data section
        .align 4
x:      .word  7
```

Getting the address of x.

The code for

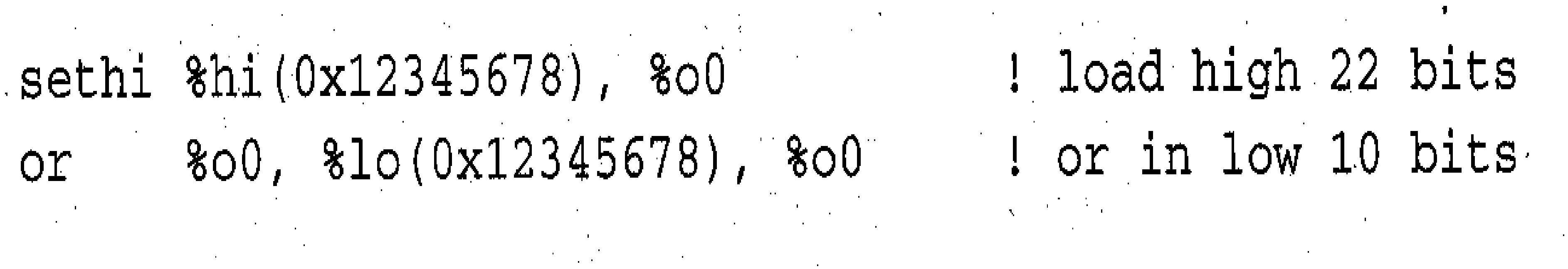


```
p = &x;
```

is

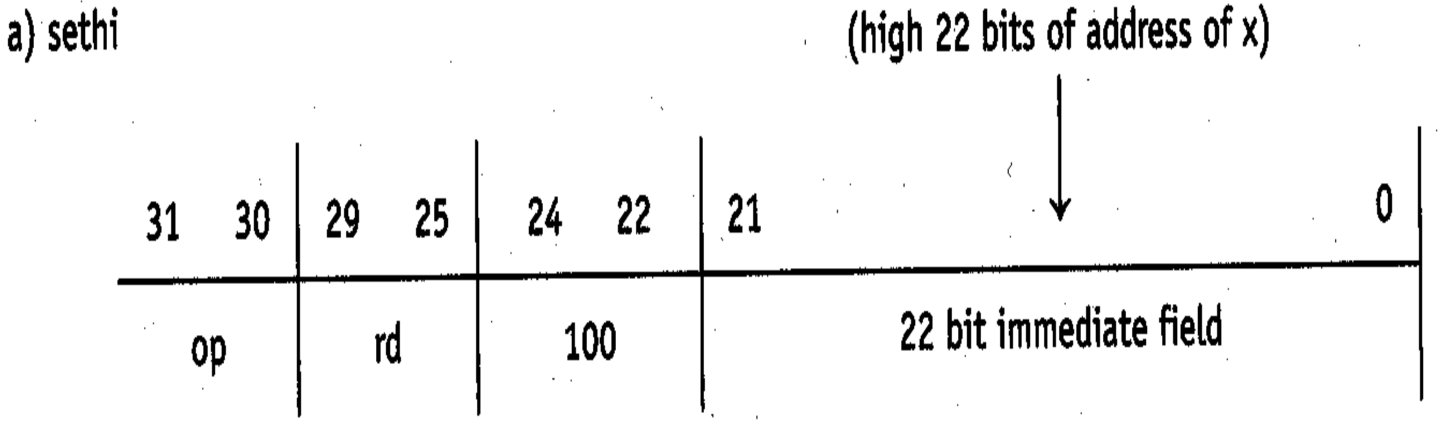
```
sethi  %hi(x),  %i0                  ! get address of x
or     %i0,  %lo(x),  %i0

sethi  %hi(p),  %i1                  ! store address in p
st     %i0, [%i1 +  %lo(p)]
```

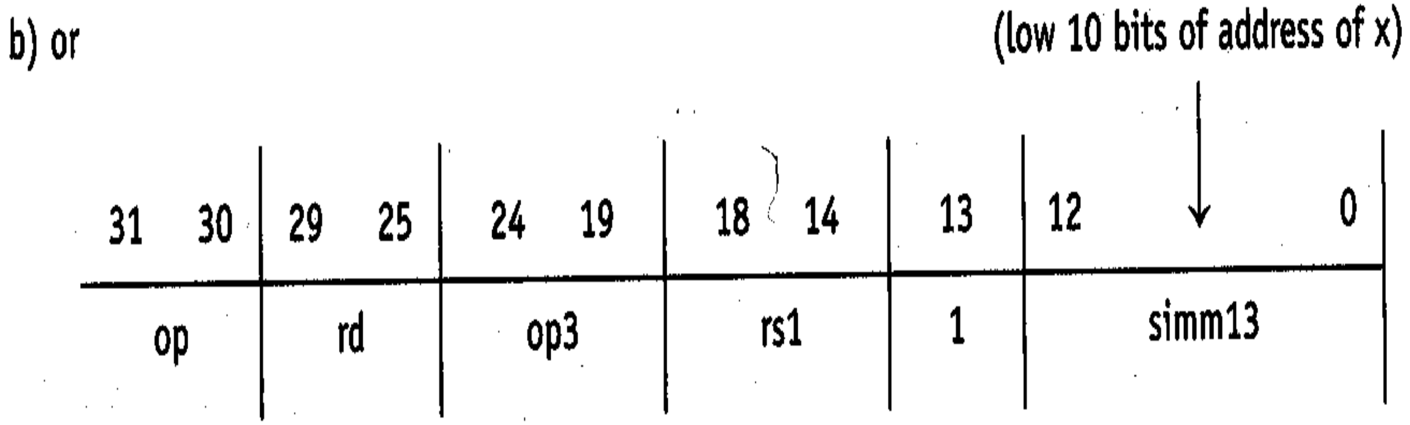
Loading a 32-bit constant



```
sethi %hi(0x12345678), %o0          ! load high 22 bits
or    %o0, %lo(0x12345678), %o0     ! or in low 10 bits
```

FIGURE 15.13


a) sethi
(high 22 bits of address of x)
31 30
29 25
24 22
21
0
op
rd
100
22 bit immediate field


b) or
(low 10 bits of address of x)
31 30
29 25
24 19
18 14
13
12
0
op
rd
op3
rs1
1
simm13

The 64-bit SPARC machines have an additional set of condition codes, xcc, that are set according to the results of 64-bit operations. icc is set according to the results of 32-bit operations.

The 64-bit SPARC machines have branch instructions that test either icc or xcc. These are the new branch instructions. The old branch instructions test only icc.

The call instruction has a 30-bit displacement fields that allows a transfer to any word address.

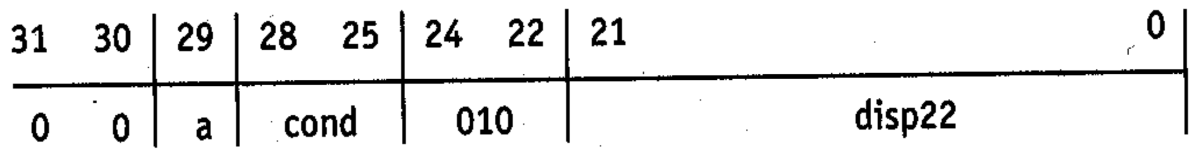
Branch instructions

```
ba    xxx              ! old, use icc
ba    %icc, xxx        ! new, use icc
ba    %xcc, xxx        ! new, use xcc
```

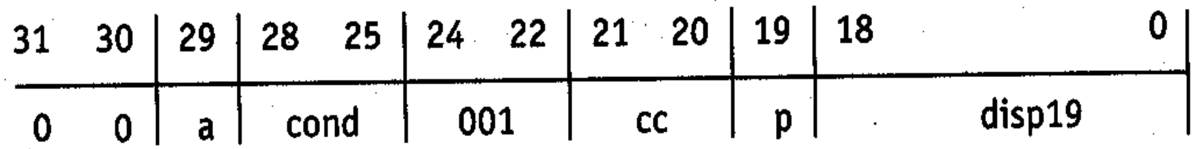
FIGURE 15.14 Branch instructions

Opcode	Cond	Mnemonic	Name (test)
00	1000	ba	Branch always
00	0000	bn	Branch never
00	0001	be	Branch equal (Z == 1)
00	0010	ble	Branch less or equal ((N ^ V) == 1 || Z == 1)
00	0100	bleu	Branch less or equal unsigned (C == 1 || Z == 1)
00	0011	bl	Branch less ((N ^ V) == 1)
00	0101	bcs	Branch carry set (C == 1)
		blu	Branch less unsigned (C == 1)
00	0110	bneg	Branch negative (N == 1)
00	1001	bne	Branch not equal (Z == 0)
00	1010	bg	Branch greater (Z == 0 && N ^ V == 0)
00	1100	bgu	Branch greater unsigned (Z == 0 && C == 0)
00	1011	bge	Branch greater or equal (N ^ V == 0)
00	1101	bcc	Branch carry clear (C == 0)
		bgeu	Branch greater or equal unsigned (C == 0)
00	1110	bpos	Branch positive (N == 0)
00	0111	bvs	Branch signed overflow (V == 1)
00	1111	bvc	Branch no signed overflow (V == 0)

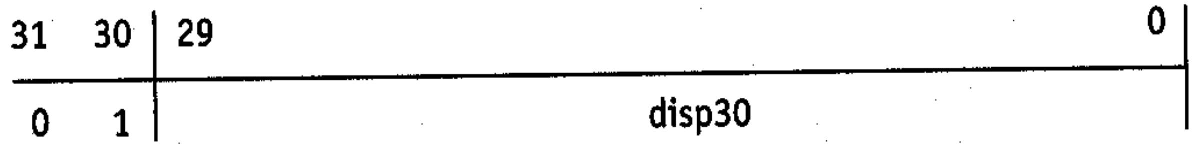
FIGURE 15.15 a) Old branch instruction



b) New branch instruction



c) Call instruction



a:	annul bit
cond:	condition
cc:	condition code set (00: icc, 10: xcc)
p:	prediction bit
disp19:	19-bit signed word displacement
disp22:	22-bit signed word displacement
disp30:	30-bit signed word displacement

Comparing using subcc

```
subcc   %i0, %i1, %g0
bl      xxx
```

Instruction pipelining

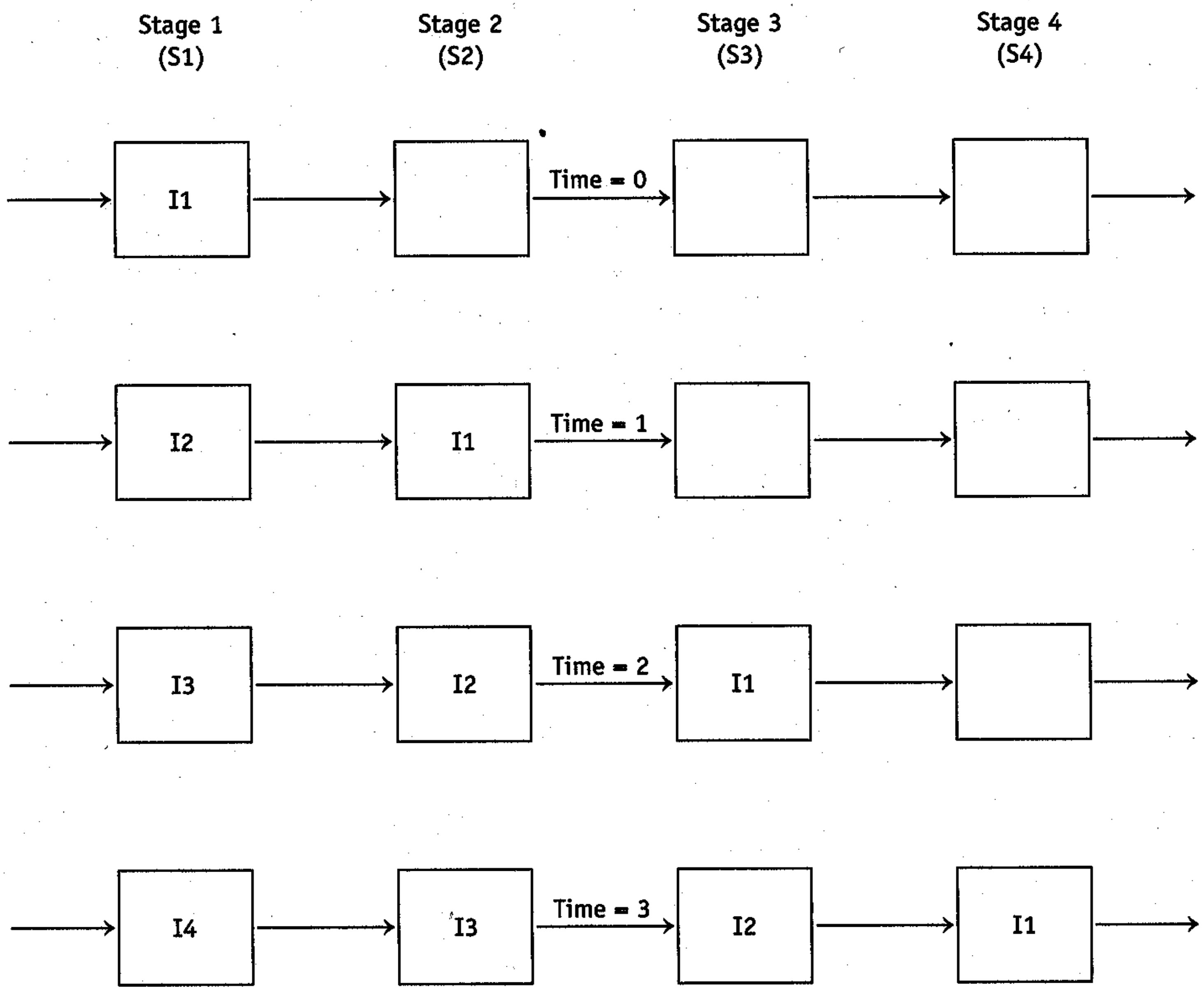
Same principle as a car assembly line. A car is built in stages. An instruction is executed in stages.

A four-stage pipeline can increase execution rate by a factor of 4.

Stages of our pipeline

- **Fetch** (fetch instruction)
- **Decode** (decode opcode, access operands, compute effective address)
- **Execute** (Perform ALU operation)
- **Write back** (Update registers)

FIGURE 15.16 a) Snapshots of a four-stage pipeline



b) Instruction vs. time representation

Time

Instruction	0	1	2	3
I1	S1	S2	S3	S4
I2		S1	S2	S3
I3			S1	S2
I4				S1

Data dependency

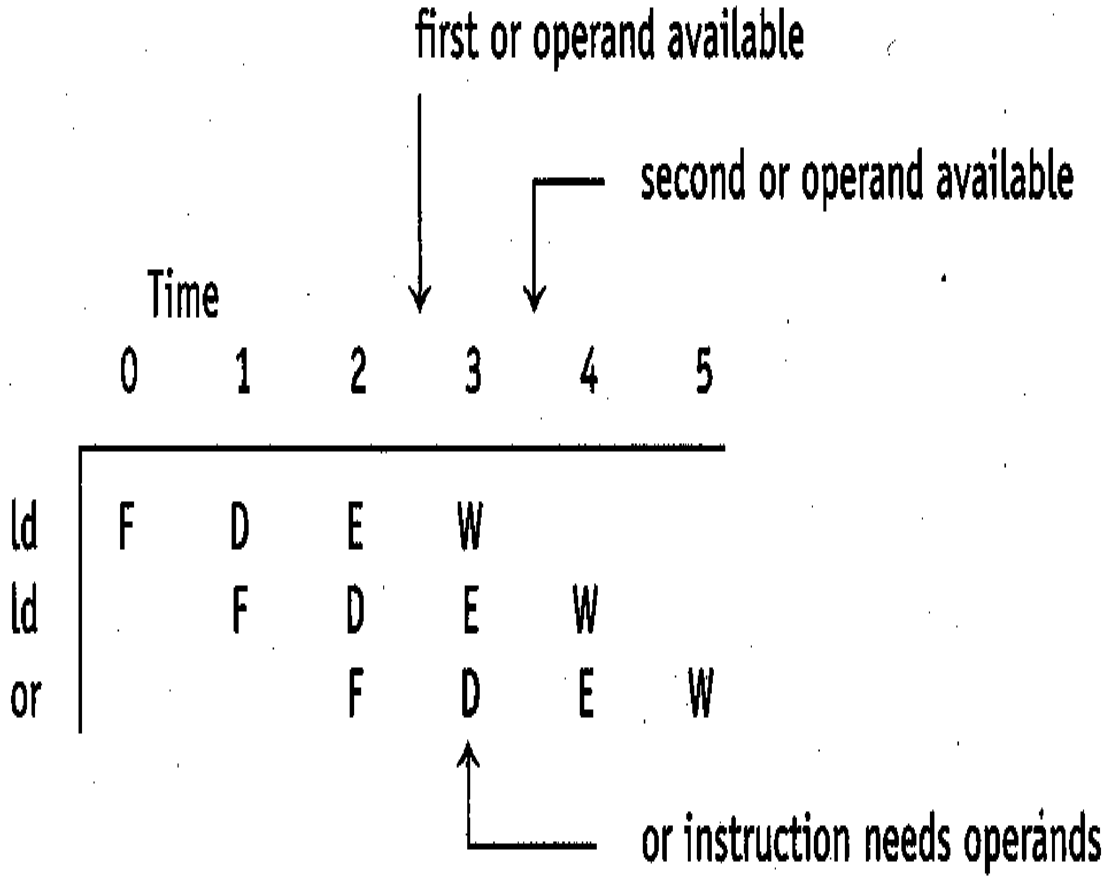
One instruction needs the data provided by another instruction

How is this sequence handled by the pipeline? Note the data dependency of the **or** on the **ld**.

```
ld    [%l0], %o0
ld    [%l1], %o1
or    %o0, %o1, %o2
```

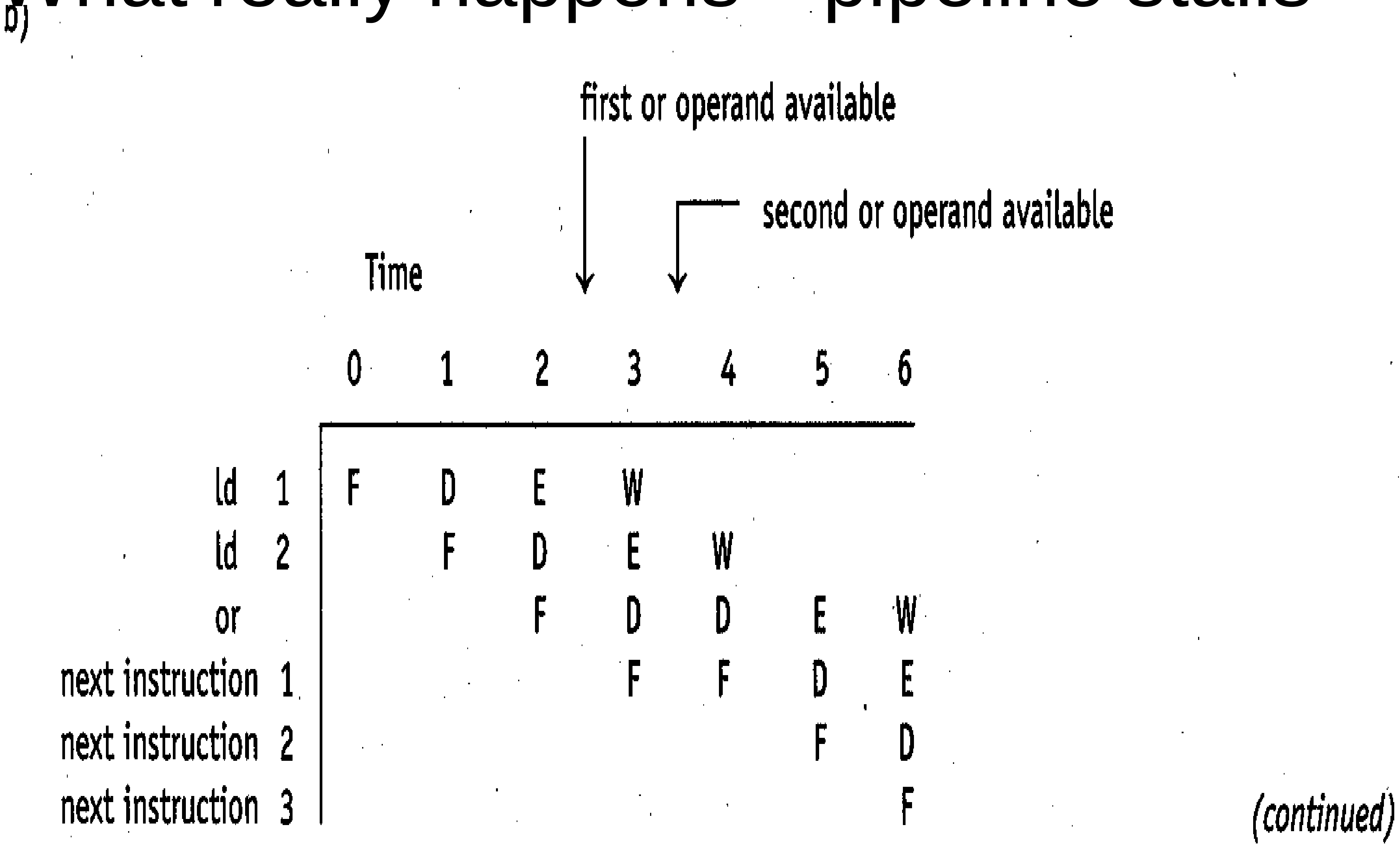
Wrong--what about the data dependency between ld and or?

FIGURE 15.17 a)


first or operand available
second or operand available
or instruction needs operands

Time	0	1	2	3	4	5
ld	F	D	E	W		
ld		F	D	E	W	
or			F	D	E	W

What really happens—pipeline stalls

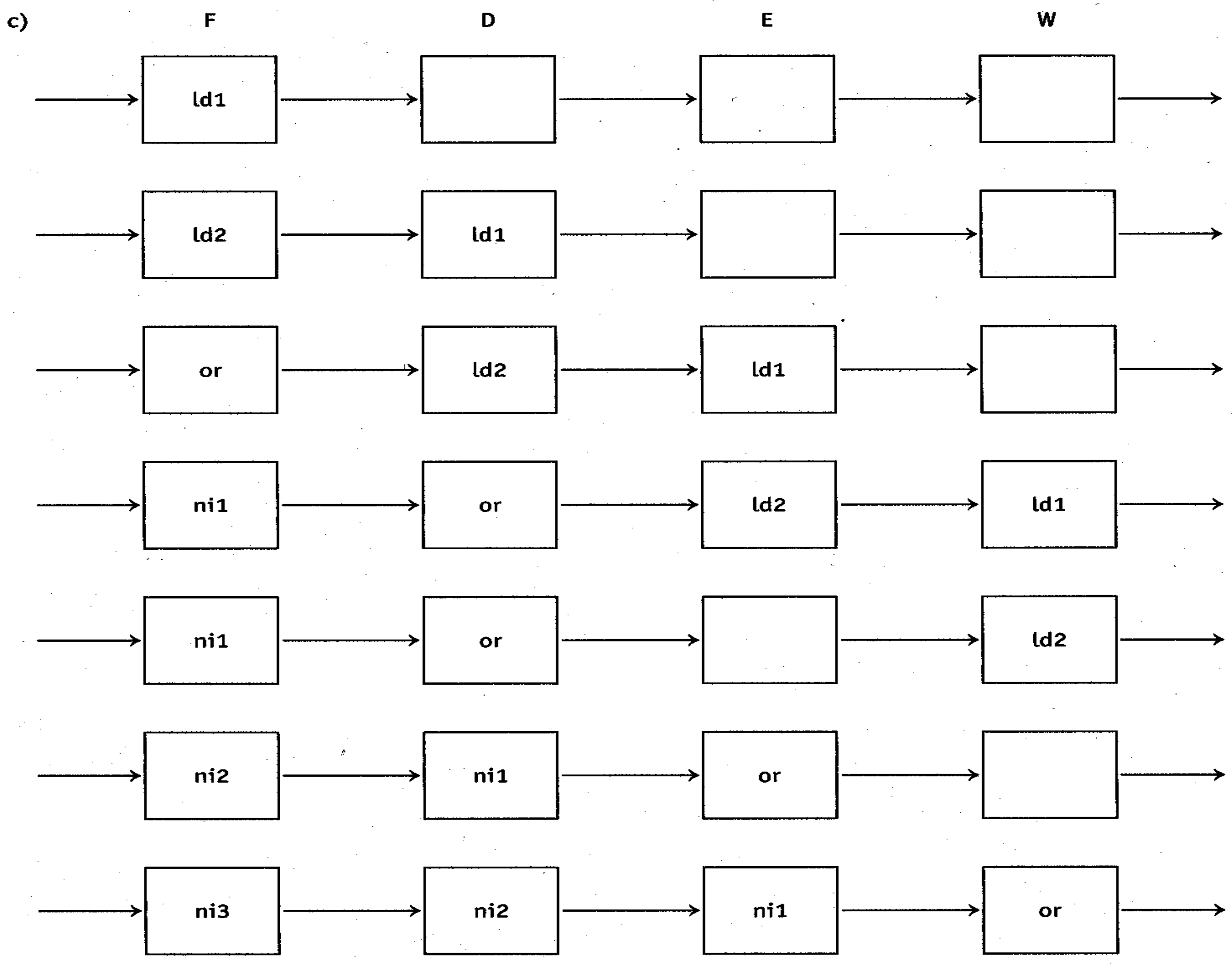

b)

	0	1	2	3	4	5	6
ld 1	F	D	E	W			
ld 2		F	D	E	W		
or			F	D	D	E	W
next instruction 1				F	F	D	E
next instruction 2						F	D
next instruction 3							F

(continued)

Bubble appears in pipeline because of stall

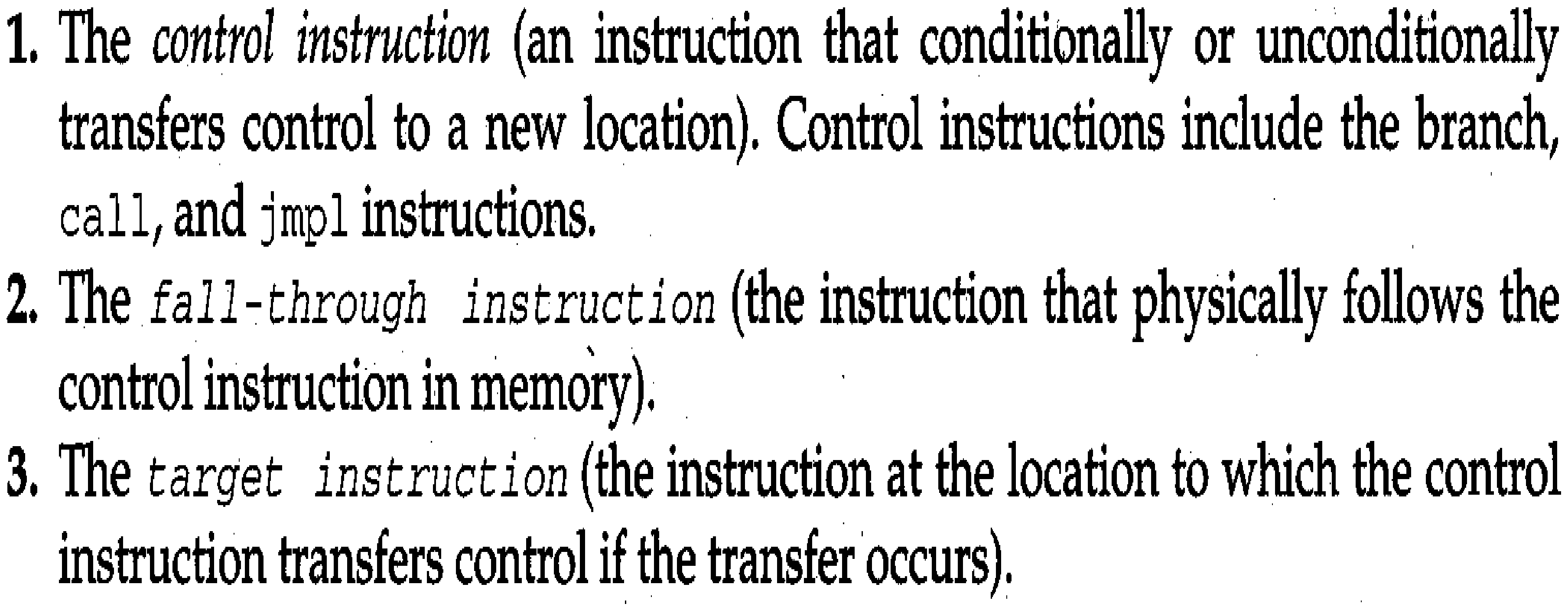
FIGURE 15.17
(continued)



Control dependency

The availability of one instruction depends on another.

Some useful terminology

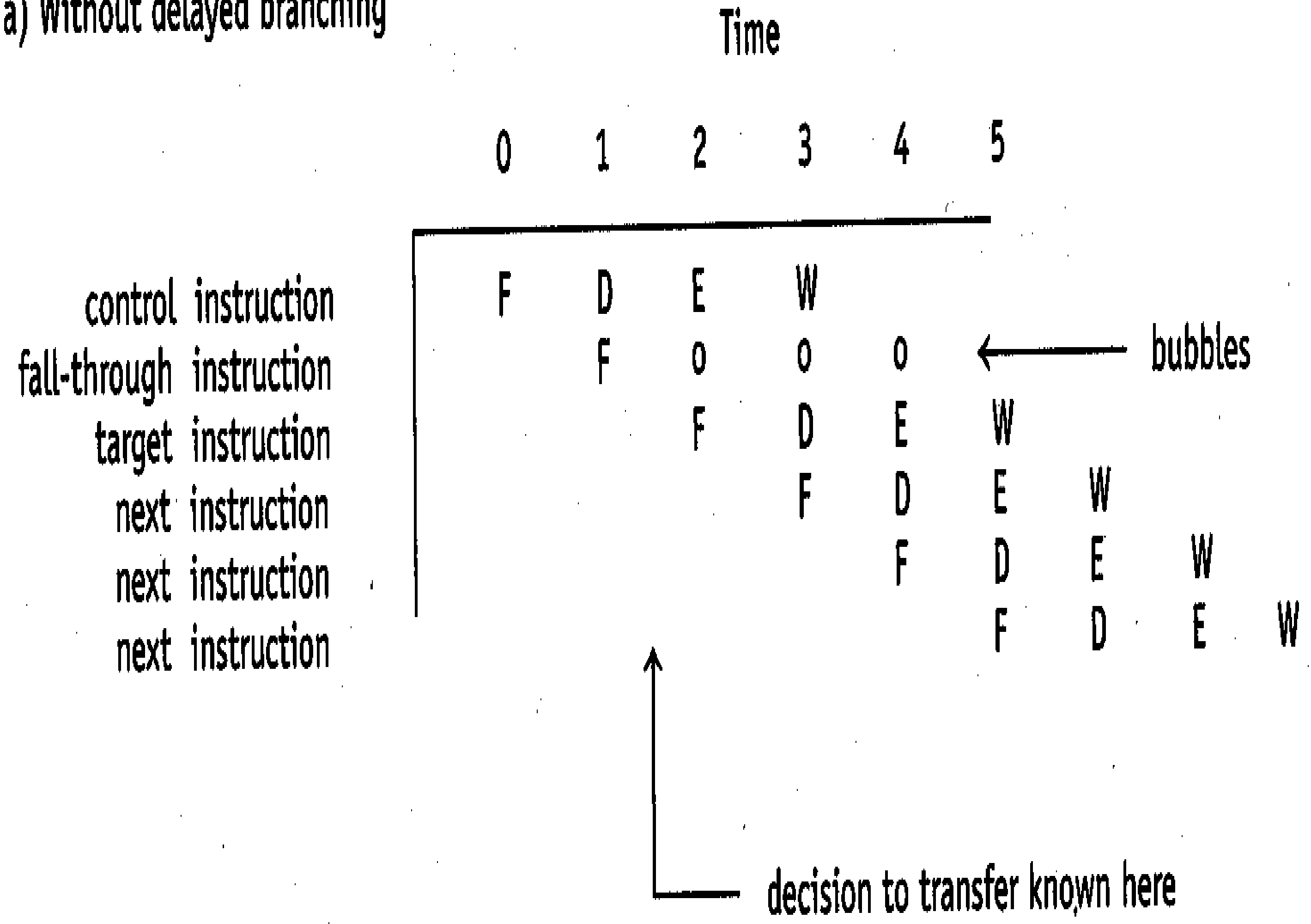


1. The *control instruction* (an instruction that conditionally or unconditionally transfers control to a new location). Control instructions include the branch, `call`, and `jmpl` instructions.
2. The *`fall-through instruction`* (the instruction that physically follows the control instruction in memory).
3. The *`target instruction`* (the instruction at the location to which the control instruction transfers control if the transfer occurs).

Delayed branching

Allow the fall-through instruction to execute whether or not the transfer of control occurs.

FIGURE 15.18 a) Without delayed branching



b) With delayed branching

Time

	0	1	2	3	4	5			
control instruction	F	D	E	W					
fall-through instruction		F	D	E	W				
target instruction			F	D	E	W			
next instruction				F	D	E	W		
next instruction					F	D	E	W	
next instruction						F	D	E	W

To pass 5 in %o0 to sub

```
call sub_ii
or   %g0, 5, %o0        ! load 5 into %o0
```

or completes before transfer of control

Only the transfer of control is delayed on delayed branching. In all other respects, the delay-slot instruction executes after the control instruction. Thus, you **cannot** reorder the first sequence to the second sequence below.

```
ld    [%l0], %o2
subcc %o0, %o1, %g0        ! compare %o0 with %o1
bl    xxx                  ! branch based on result of subcc
```

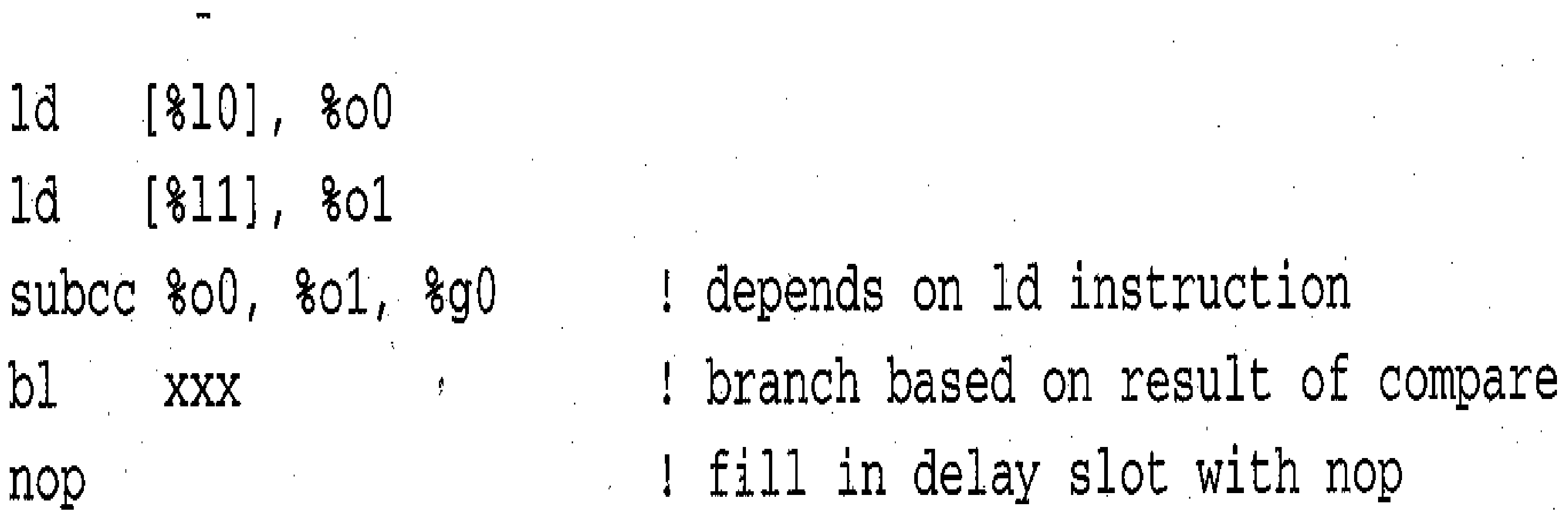
to

```
ld    [%i0], %o2
bl    xxx                  ! branch based on result of subcc
subcc %o0, %o1, %g0        ! compare %o0 with %o1
```

But the reordering below is ok because the bl instruction does not depend on the ld instruction.

```
subcc %o0, %o1, %g0       ! compare %o0 with %o1
bl    xxx                 ! branch based on result of subcc
ld    [%i0], %o2
```

If you can’t move any instruction into the delay slot, use nop



```
ld    [%l0], %o0
ld    [%l1], %o1
subcc %o0, %o1, %g0       ! depends on ld instruction
bl     xxx                ! branch based on result of compare
nop                       ! fill in delay slot with nop
```

Calling sequence corresponding to sub(1,2);

```
sub(1,2);
```

is

```
or %g0, 1, %o0    ! load 1 into %o0
call sub_ii
or %g0, 2, %o1    ! load 2 into %o1
```

When the call instruction is executed, it places its own address in %o7, which becomes %i7 in the called function. Thus, the return address is given by

```
%i7 + 8
```

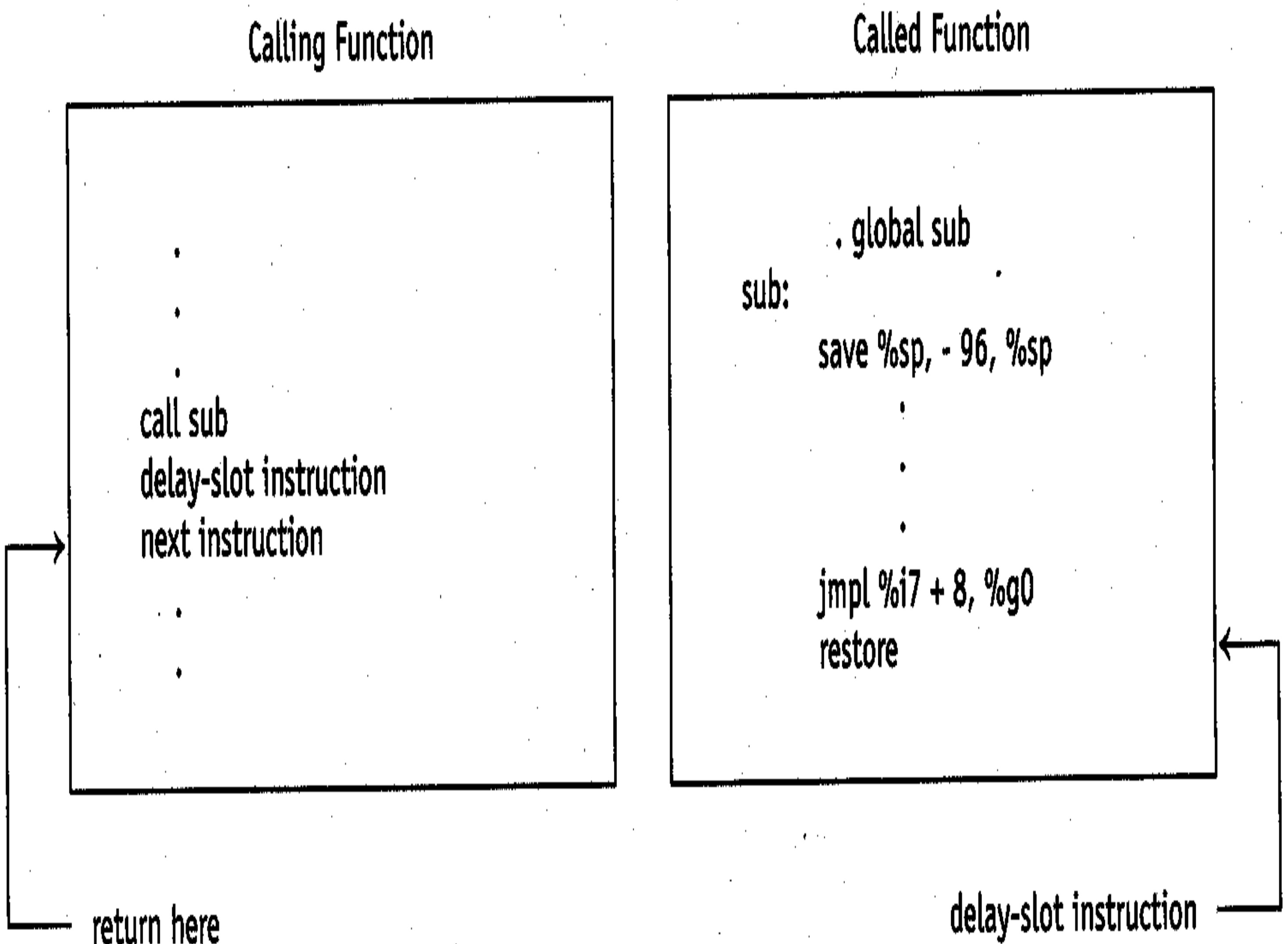
The “+ 8” is needed to skip over the call and delay slot instructions. The instruction

```
jmpl  %i7 + 8, %g0
```

in the called function jumps to the return address.

Linkage instructions

FIGURE 15.19



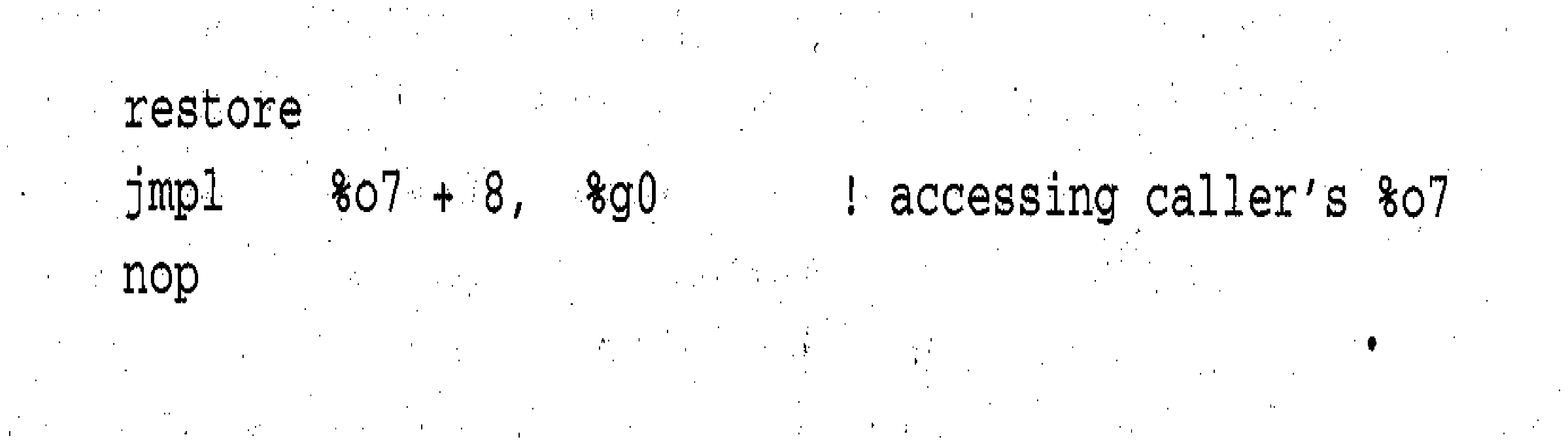
```
call sub
delay-slot instruction
next instruction
```

```
        .global sub
sub:
        save %sp, - 96, %sp
        .
        .
        .
        jmpl %i7 + 8, %g0
        restore
```

You should not use %i7 as shown below because the register switch has already occurred when the jmpl instruction is executed.

```
restore
jmpl    %i7 + 8,  %g0       ! accessing caller's %i7
nop
```

Correct return sequences



```
restore
jmpl    %o7 + 8,  %g0        ! accessing caller's %o7
nop
```

or

```
jmpl    %i7 + 8, %g0
restore    ; switch after jmpl so use %i7
```

Addressing modes

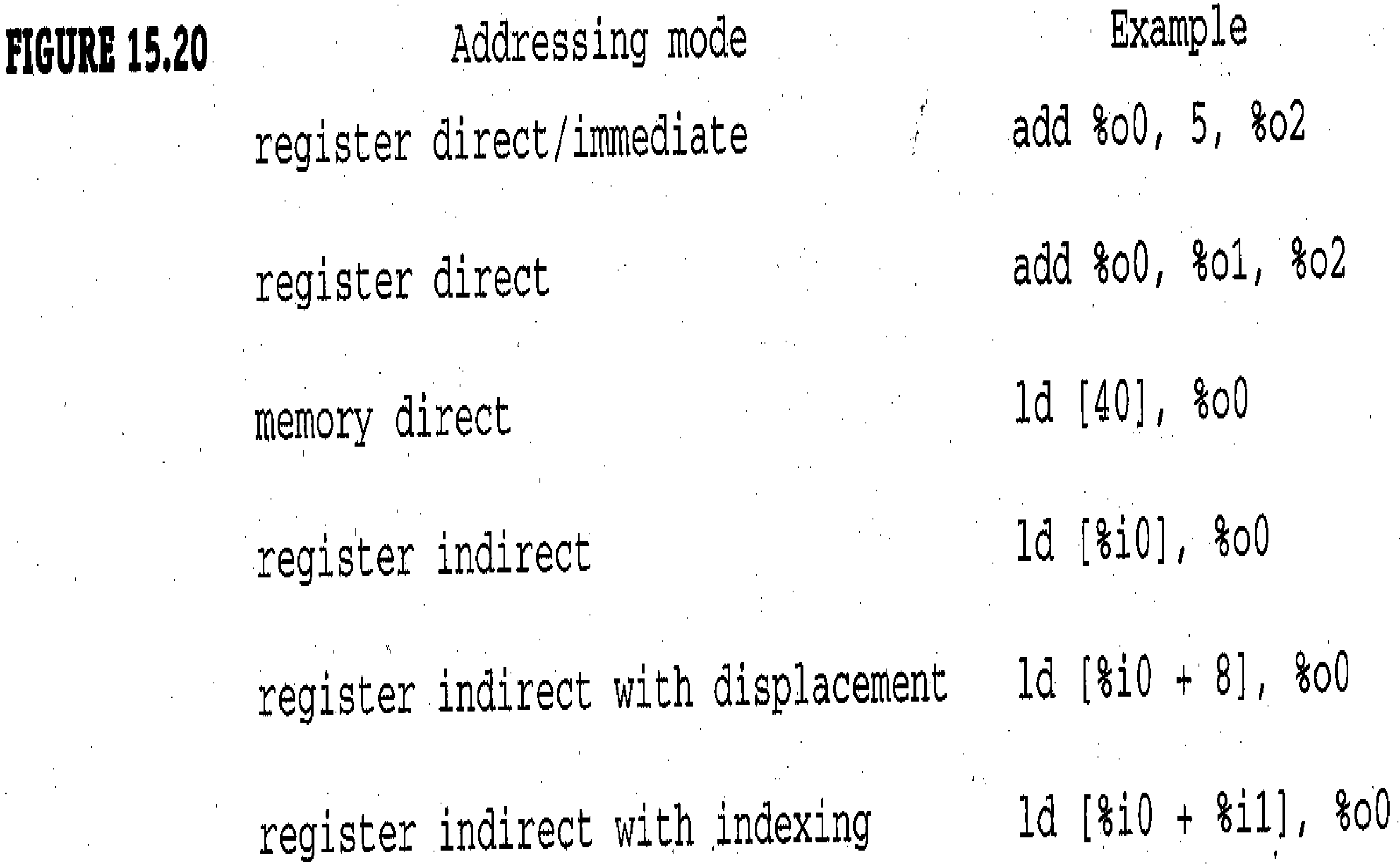


FIGURE 15.20

Addressing mode	Example
register direct/immediate	add %o0, 5, %o2
register direct	add %o0, %o1, %o2
memory direct	ld [40], %o0
register indirect	ld [%i0], %o0
register indirect with displacement	ld [%i0 + 8], %o0
register indirect with indexing	ld [%i0 + %i1], %o0

Using an index register

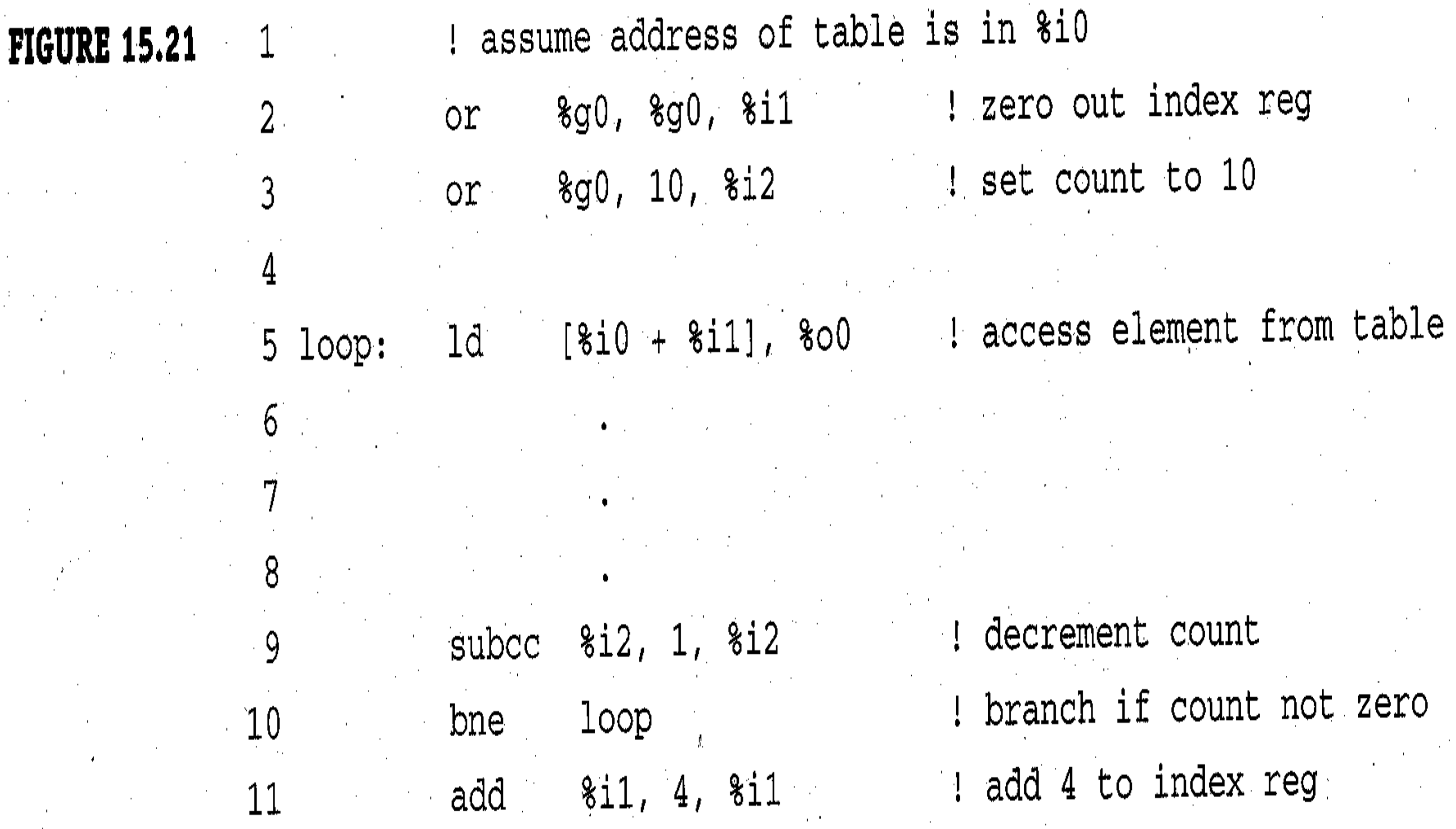
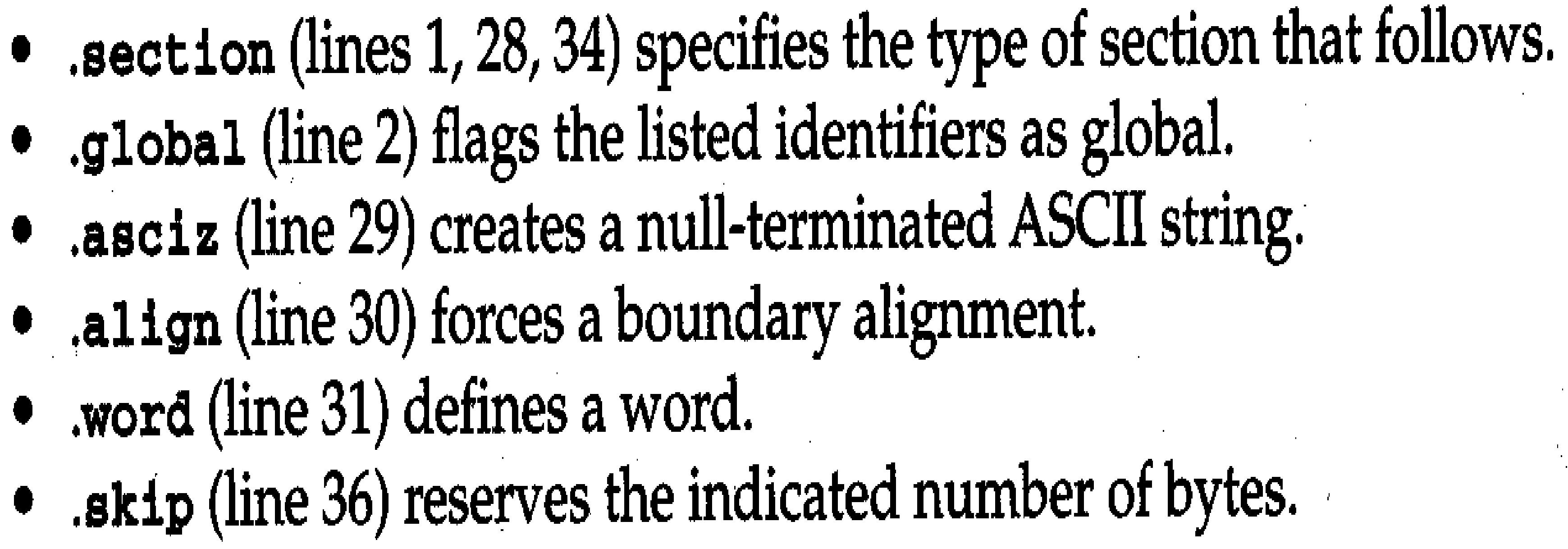


FIGURE 15.21

```
        ! assume address of table is in %i0
        or     %g0, %g0, %i1        ! zero out index reg
        or     %g0, 10, %i2         ! set count to 10

loop:   ld     [%i0 + %i1], %o0     ! access element from table
                 .
                 .
                 .
        subcc  %i2, 1, %i2          ! decrement count
        bne    loop                 ! branch if count not zero
        add    %i1, 4, %i1          ! add 4 to index reg
```

Assembler directives



- **.section** (lines 1, 28, 34) specifies the type of section that follows.
- **.global** (line 2) flags the listed identifiers as global.
- **.asciz** (line 29) creates a null-terminated ASCII string.
- **.align** (line 30) forces a boundary alignment.
- **.word** (line 31) defines a word.
- **.skip** (line 36) reserves the indicated number of bytes.

The program on the next slide illustrates the use of these directives.

FIGURE 15.22

```
          .section ".text"                    ! text section
          .global main
main:
          save   %sp, -96, %sp                ! create called functions frame
                                              ! and switch regs

          sethi  %hi(x), %l7                  ! get x
          ld [%l7 + %lo(x)], %o0

          sethi  %hi(y), %l7                  ! get y
          ld [%l7 + %lo(y)], %o1

          add    %o0, %o1, %o0                ! add x and y

          sethi  %hi(sum), %l7                ! store result in sum
          st     %o0, [%l7 + %lo(sum)]

          sethi  %hi(cs), %o0                 ! get high part of address of cs
          or     %o0, %lo(cs), %o0            ! get low part of address of cs

          sethi %hi(sum), %l7                 !get high part of address of sum
          call printf
          ld [%l7 + %lo(sum)],%o1             ! load sum into % 01

          jmpl   %i7 + 8, %g0                 ! return to caller
          restore                             ! switch to caller's regs
!=====================================================================
          .section ".data"                    ! initialized data section
cs:       .asciz    "sum = %d\n"              ! null-terminated string
          .align    4                         ! force full-word boundary
x:        .word     1                         ! integer word
y:        .word     15                        ! integer word
!=====================================================================
          .section ".bss"                     ! uninitialized data section
          .align    4
sum:      .skip     4
```

Termination sequence (an alternative to jmpl-restore)

```
or   %g0, 1, %g1          ! load 1 (terminate program) into %g1
ta   0                    ! trap to operating system
```

A *synthetic instruction* is shorthand for another instruction. For example,

```
    cmp   %i0, %i1
```

is the synthetic instruction for

```
    subcc  %o0, %i1, %g0
```

FIGURE 15.23

Synthetic Instruction	Real Instruction	Function
`btst reg1/immed, reg2`	`andcc reg1, reg2/immed, %g0`	bit test
`bset reg1/immed, reg2`	`or reg1, reg2/immed, reg1`	bit set
`bclr reg1/immed, reg2`	`andn reg1, reg2/immed, reg1`	bit clear
`btog reg1/immed, reg2`	`xor reg1, reg2/immed, reg1`	bit toggle
`clr reg`	`or %g0, %g0, reg`	clear
`clr [address]`	`st %g0, [address]`	clear
`clrh [address]`	`sth %g0, [address]`	clear half
`clrb [address]`	`stb %g0, [address]`	clear byte
`cmp reg1, reg2/immed`	`subcc reg1, reg2/immed, %g0`	compare
`dec reg`	`sub reg, 1, reg`	decrement
`dec immed, reg`	`sub reg, immed, reg`	decrement
`deccc reg`	`sub reg, 1, reg`	decrement
`deccc immed, reg`	`subcc reg, immed, reg`	decrement
`inc reg`	`add reg, 1, reg`	increment
`inc immed, reg`	`add reg, immed, reg`	increment
`inccc reg`	`addcc reg, 1, reg`	increment
`inccc immed, reg`	`addcc reg, immed, reg`	increment
`mov reg1/immed, reg2`	`or %g0, reg1/immed, reg2`	move
`neg reg`	`sub %g0, reg, reg`	negate
`nop`	`sethi 0, %g0`	no operation
`ret`	`jmpl %i7 + 8, %g0`	return
`retl`	`jmpl %o7 + 8, %g0`	ret from leaf
If 4096 <= value < 4096		
`set value, reg`	`or %g0, value, reg`	set to value
If value & 0x3ff == 0		
`set value, reg`	`sethi %hi(value), reg`	set to value
If value neither		
`set value, reg`	`sethi %hi(value), reg`	set to value
	`or reg, %lo(value), reg`	

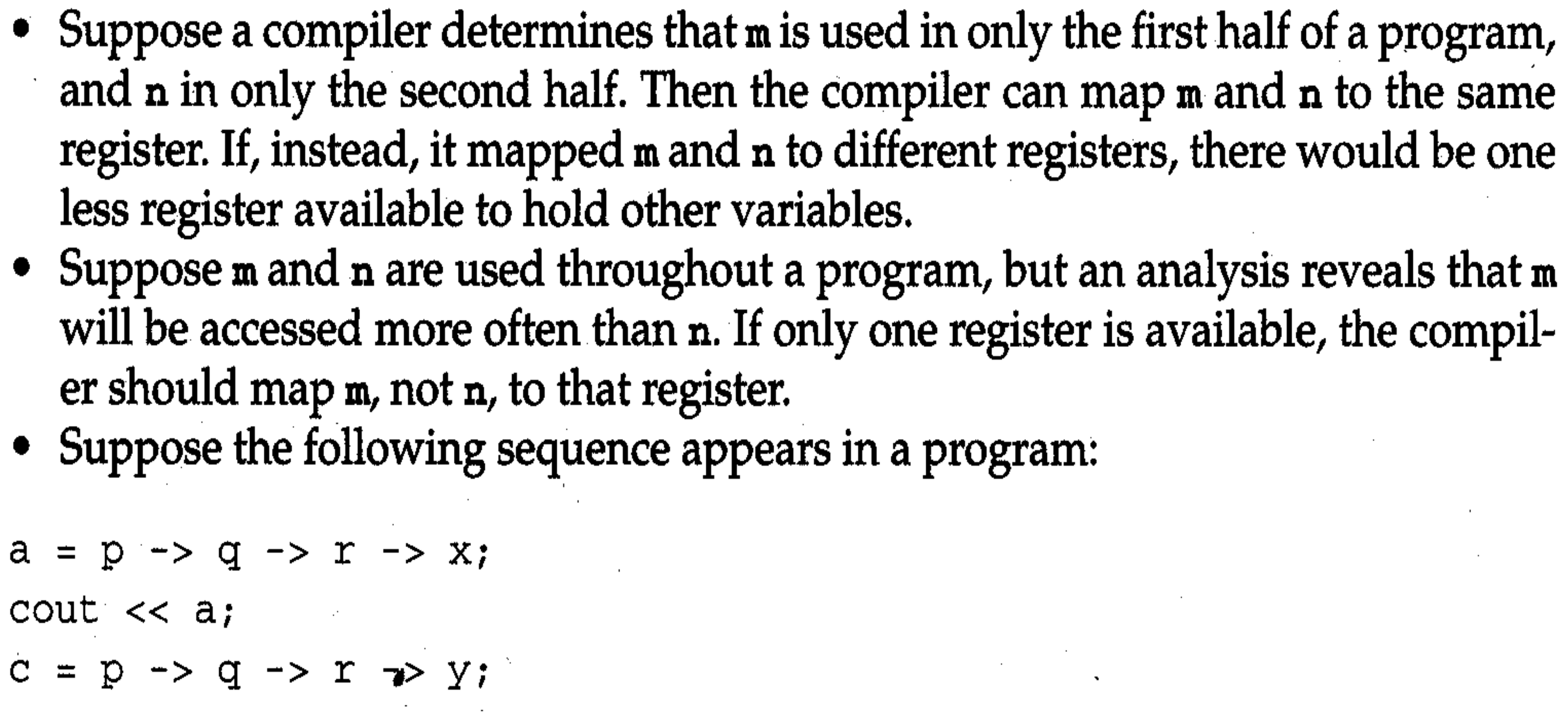
Be careful how you use synthetic instructions

```
call sub
set  0x12345678, %o0
```

This sequence, in reality, is

```
call sub
sethi %hi(0x12345678), %o0
or    %o0, %lo(0x12345678), %o0
```

Structural analysis is important for code optimization



- Suppose a compiler determines that **m** is used in only the first half of a program, and **n** in only the second half. Then the compiler can map **m** and **n** to the same register. If, instead, it mapped **m** and **n** to different registers, there would be one less register available to hold other variables.
- Suppose **m** and **n** are used throughout a program, but an analysis reveals that **m** will be accessed more often than **n**. If only one register is available, the compiler should map **m**, not **n**, to that register.
- Suppose the following sequence appears in a program:

```
a = p -> q -> r -> x;
cout << a;
c = p -> q -> r -> y;
```

Does p -> q -> r have to be computed twice? No, in this case.

Examples of C++ programs and their compiler-generated SPARC assembly code follow.

FIGURE 15.24 a) C++ code

```
int gv1, gv2 = 5;
int fa(int x, int y, int z)
{
  return x + y + z;
}
int main()
{
  int lv1, lv2 = 7;

  lv1 = 11;
  gv1 = fa(gv2, lv1, lv2);

  return 0;
}
```

FIGURE 15.24 (continued)

b) SPARC code

```
                .section ".text"
                .global   fa_iii

 fa_iii:        add       %o0,  %o1,  %o0        ! return x + y + z;
                retl
                add       %o0,  %o2,  %o0

                .global   main
 main:          save      %sp, -96, %sp

                sethi     %hi(gv2), %o0          ! gv1 = fa(gv2, lv1, lv2);
                ld        [%o0+%lo(gv2)], %o0
                mov       11, %o1                ! lv1 = 11;
                call      fa_iii
                mov       7, %o2                 ! int lv2= 7;
                sethi     %hi(gv1), %o1          ! store ret value in gv1
                st        %o0,[%o1 + %lo(gv1)]

                ret
                restore   %g0, 0, %o0
 !=========================================================
                .section ".data"
                .global   gv2                    ! int gv2 = 5;
                .align    4
 gv2:           .word     5
 !=========================================================
                .section ".bss"
                .global   gv1
                .align    4
 gv1:           .skip     4
```

FIGURE 15.25

a)

```
int gv;
int fb(int t, int u, int v, int w, int x, int y, int z)
{
    return t + u + v + w + x + y + z;
}
int main()
{
    gv = fb(1, 2, 3, 4, 5, 6, 7);
    return 0;
}
```

b)

```
 1                 .section ".text"
 2                 .global  fb_iiiiiii
 3 fb_iiiiiii:
 5                 add     %o0,  %o1,  %o0    !return t + u + v + w + x + y + z;
 6                 add     %o0,  %o2,  %o0
 7                 add     %o0,  %o3,  %o0
 8                 add     %o0,  %o4,  %o0
 9                 add     %o0,  %o5,  %o0
 4                 ld      [%sp+92],   %g1
10                 retl
11                 add     %o0, %g1,   %o0
12
13                 .global main
14 main:           save    %sp, -96, %sp
15
16                 mov     1, %o0             ! gv = g(1, 2, 3, 4, 5, 6, 7);
17                 mov     2, %o1
18                 mov     3, %o2
19                 mov     4, %o3
20                 mov     5, %o4
21                 mov     6, %o5
22                 mov     7, %g1
23                 call    fb_iiiiiii
24                 st      %g1, [%sp+92]
25
26                 ret                        ! return 0;
27                 restore %g0, 0, %o0
```

FIGURE 15.26 a)

```
void fc(int *p)
{
    *p = 99;
}
int main()
{
    int lv;
    fc(&lv);
    return 0;
}
```

b)

```
                .section ".text"
                .global fc_pi
fc_pi:          mov     99, %g1           ! *p = 99;
                retl
                st      %g1, [%o0]

                .global main
main:           save    %sp, -96, %sp

                call    fc_pi             !  fc(&lv);
                add     %sp, 92, %o0

                ret                       ! return 0;
                restore %g0, 0, %o0
```

FIGURE 15.27

a)

```
int *gpv;
void fnull()
{}
void fd(int x)
{
    gpv = &x;
    *gpv = x + 3;
    fnull();             // forces fd to save/restore
}
int main()
{
    int lv = 7;

    fd(lv);

    return 0;
}
```

b)

```
                .section ".text"
                .global   fnull_v
fnull_v:        retl
                nop

                .global   fd_i
fd_i:           save      %sp, -96, %sp

                sethi     %hi(gpv), %o1          ! gpv = &x;
                add       %fp, 68, %o0
                st        %o0, [%o1+%lo(gpv)]

                mov       %i0, %o0               ! *gpv = x + 3;
                add       %o0, 3, %o0
                call      fnull_v                ! fnull();
                st        %o0, [%fp+68]

                ret
                restore

                .global   main
main:           save      %sp, -96, %sp

```

(continued)

FIGURE 15.27 (continued)

```
             call     fd_i                 ! fd(lv);
             mov      7, %o0

             ret
             restore
!=====================================================================
             .section ".bss"
             .global  gpv
             .align   4
gpv:         .skip    4
```

Memory-mapped I/O on SPARC

```
set   0xffffff0, %o0
mov   1, %i0
stb   %i0, [%o0]          ! send read command
```

To get status, we would use

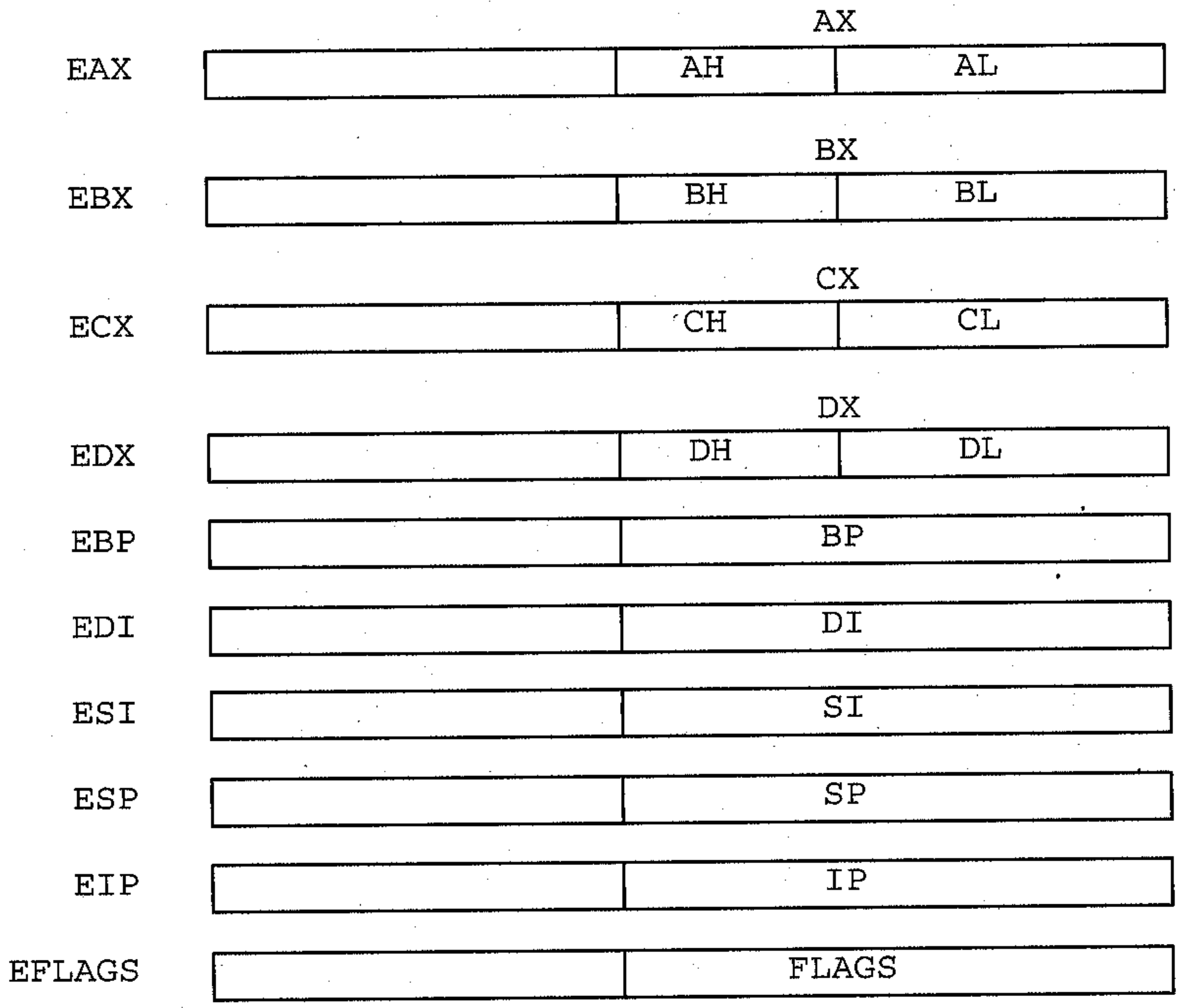
```
ldb [%o0 + 1], %i0        ! get status
```

When the status byte indicates that data is available, we would then read the data with

```
ldb  [%o0 + 2], %i0       ! get data
```

The Pentium is a CISC architecture

FIGURE 15.28 Principal Registers on the Pentium



mov instructions
Direction is right to left

```
mov    eax, 3              ; load eax with 4-byte integer 3
mov    ax, 3               ; load ax with 2-byte integer 3
mov    eax, [ebp-8]        ; load eax from memory
mov    [ebp-8], eax        ; store eax into memory
```

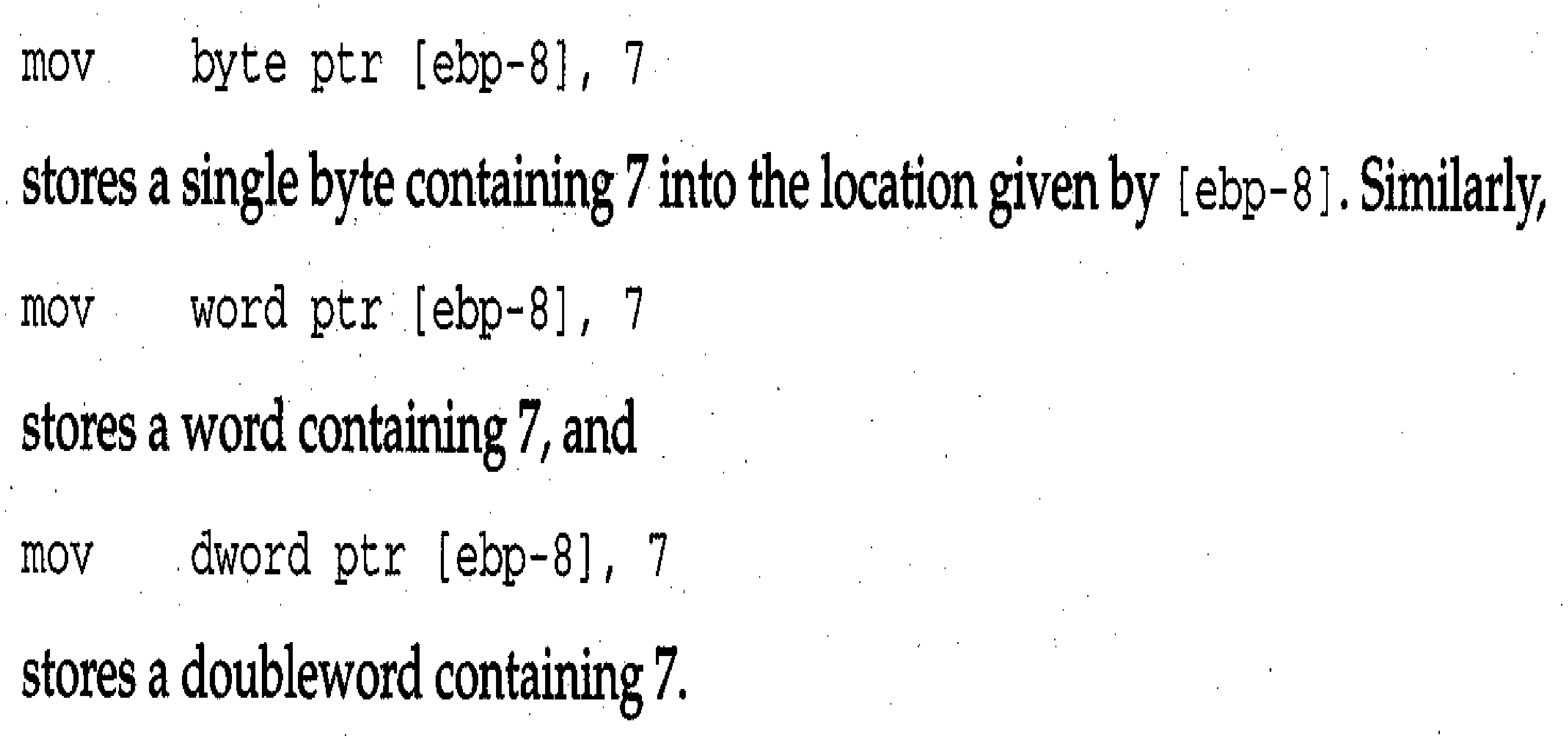
Length of target field is obvious in these instructions.

```
mov    [ebp-8], eax          ; store 4 bytes
mov    [ebp-8], ax           ; store 2 bytes
```

Length of target field is not obvious in this instruction.
Must disambiguate by specifying the length of the target field.

```
mov     [ebp-8], 7
```

How to disambiguate



```
mov     byte ptr [ebp-8], 7
```

stores a single byte containing 7 into the location given by `[ebp-8]`. Similarly,

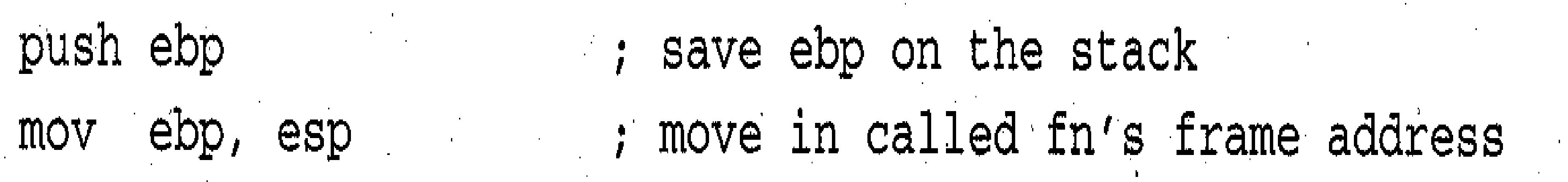
```
mov     word ptr [ebp-8], 7
```

stores a word containing 7, and

```
mov     dword ptr [ebp-8], 7
```

stores a doubleword containing 7.

push-mov sequence just like esba instruction in H1



```
push ebp                  ; save ebp on the stack
mov  ebp, esp             ; move in called fn's frame address
```

mov-pop sequence just like reba in H1

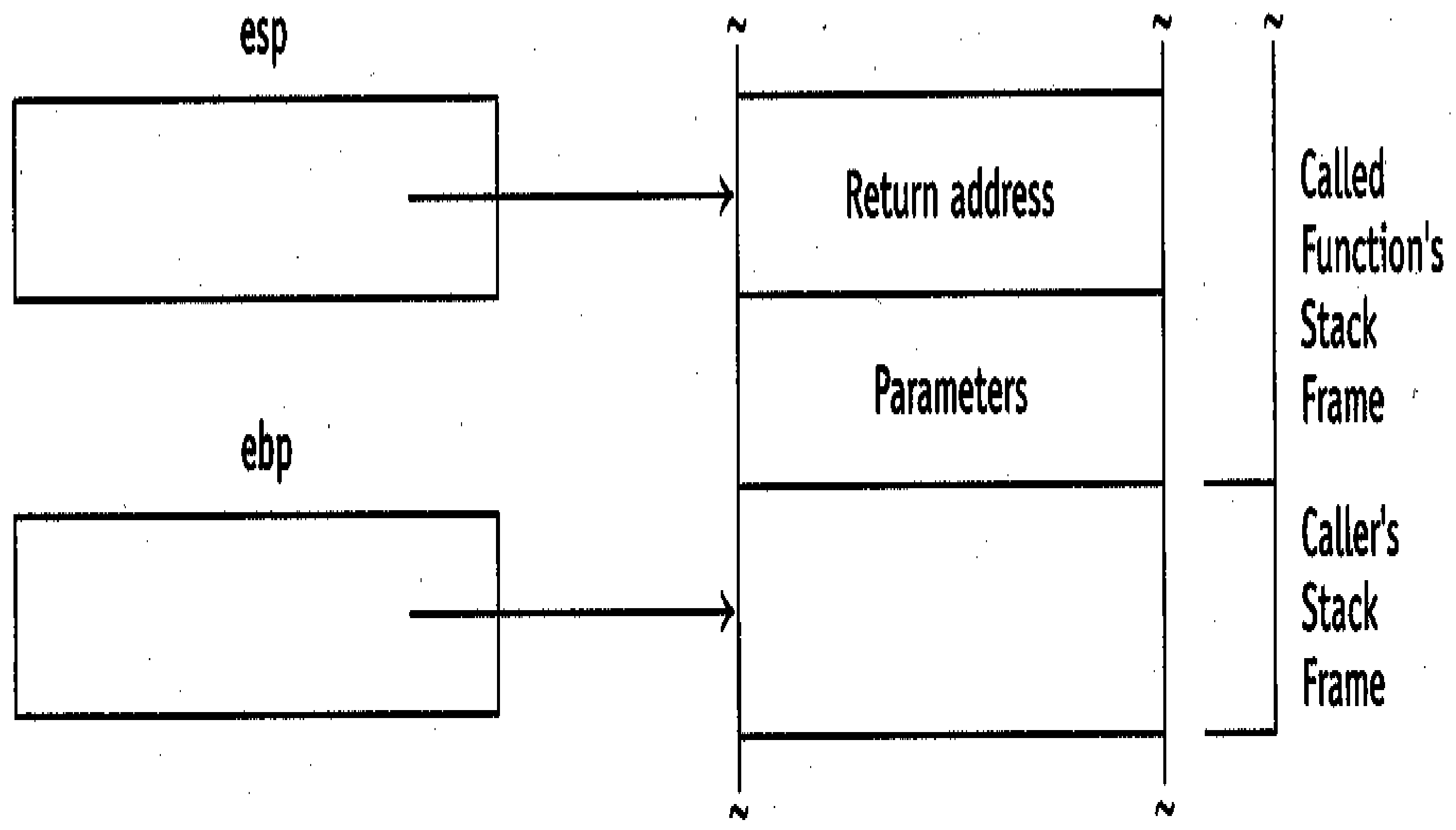
```
mov    esp, ebp
pop    ebp
```

FIGURE 15.29

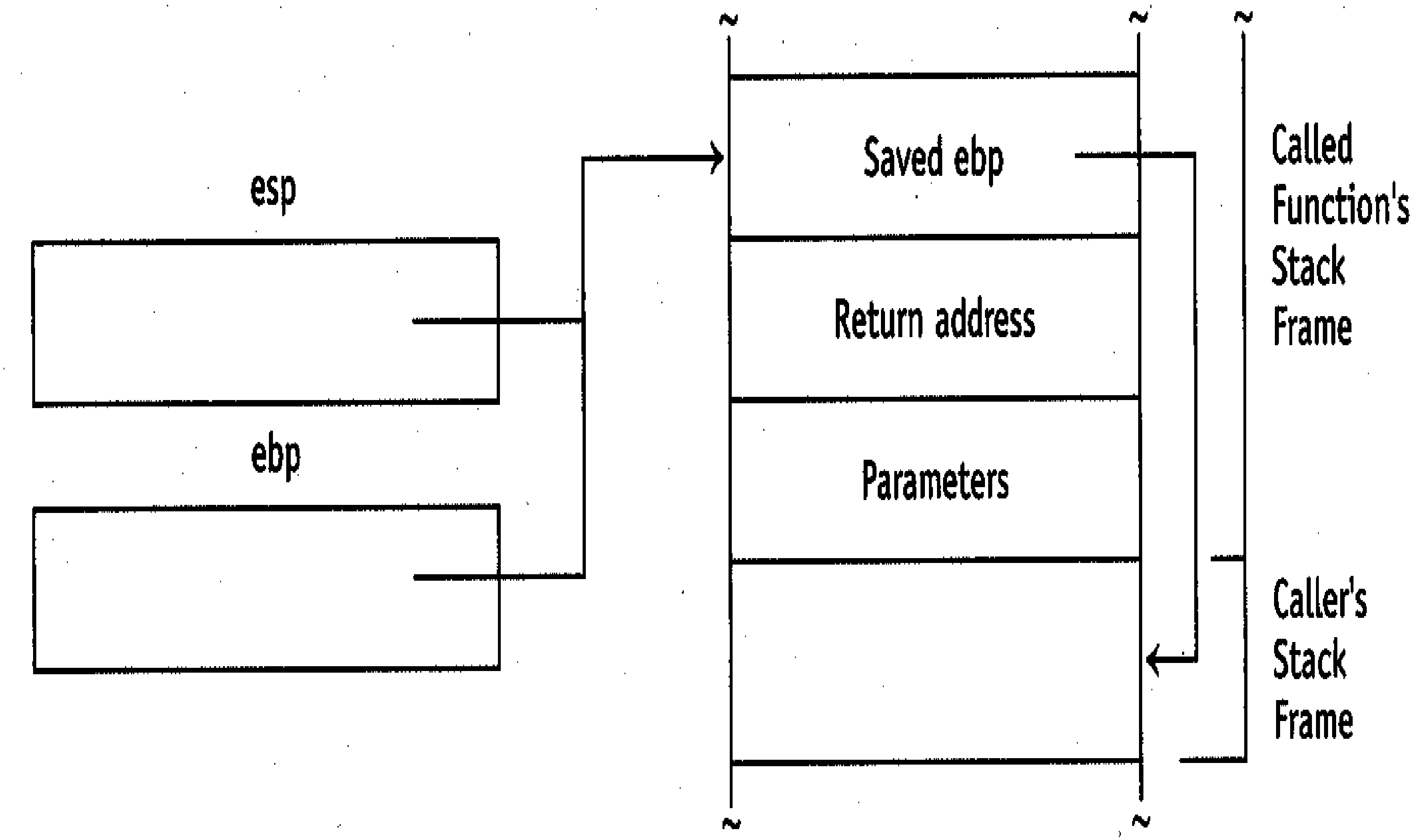
Intel Pentium code

```
.code
                public @fa$iii
@fa$iii:        push    ebp
                mov     ebp, esp

                mov     eax, [ebp+8]                ; return x + y + z;
                add     eax, [ebp+12]
                add     eax, [ebp+16]

                pop     ebp
                ret

                public main
main:           push    ebp
                mov     ebp, esp

                add     esp,-8                      ; int lv1, lv2 = 7;
                mov     dword ptr [ebp-8], 7

                mov     dword ptr [ebp-4], 11 ; lv1 = 11;

                push    dword ptr [ebp-8]           ; gv1 = fa(gv2, lv1, lv2);
                push    dword ptr [ebp-4]
                push    dword ptr [gv2]
                call    @fa$iii
                add     esp,12
                mov     [gv1], eax

                xor     eax, eax                    ; return 0;
                mov     esp, ebp
                pop     ebp
                ret
;================================================================
.data
                public gv1
gv1             dd      ?                           ; reserve 4 bytes
                public gv2
gv2             dd      5                           ; define constant 5
```

FIGURE 15.30 a) Before push/mov sequence (right after call instruction)



b) After push/mov sequence



Use addresses relative to ebp to access parameters and local variables. For example,

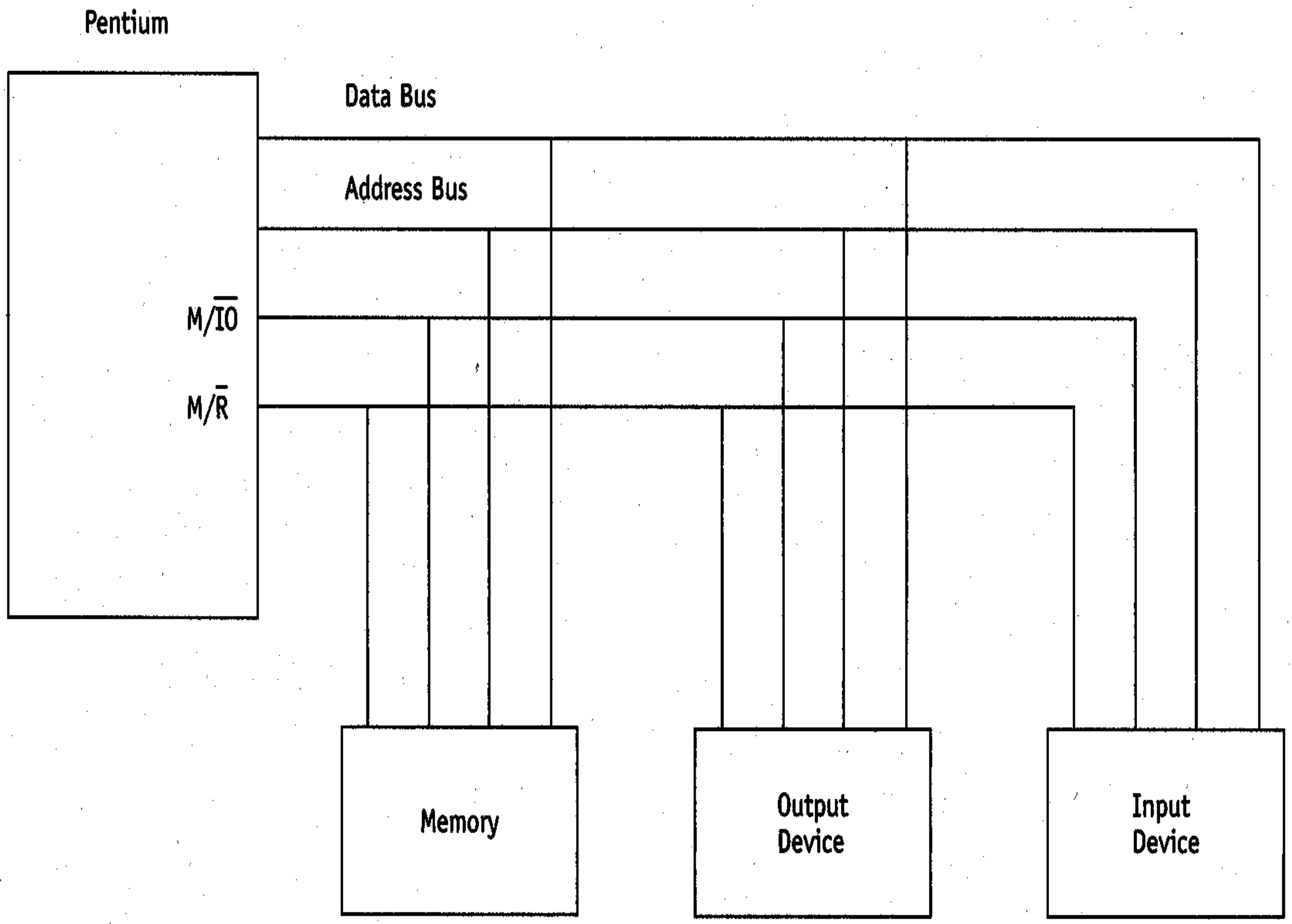
```
    mov     eax,  [ebp+8]
```

The Pentium uses I/O instructions rather than memory-mapped I/O.

FIGURE 15.31 I/O Instructions on the Pentium

```
IN      AL,DX
IN      AX,DX
IN      EAX,DX
IN      AL,<8-bit port number>
IN      AX,<8-bit port number>
IN      EAX,<8-bit port number>

OUT     DX,AL
OUT     DX,AX
OUT     DX,EAX
OUT     <8-bit port number>,AL
OUT     <8-bit port number>,AX
OUT     <8-bit port number>,EAX
```

FIGURE 15.32


Pentium
Data Bus
Address Bus
$M/\overline{IO}$
$M/\overline{R}$
Memory
Output Device
Input Device

```
IN AL,DX      ; read one byte
```

reads 1 byte (because AL is a 1-byte register) from the I/O device whose *port number* (i.e., address) is in DX, but

```
IN AX,DX      ; read two bytes
```

reads 2 bytes (because **AX** is a 2-byte register). I/O instructions can specify the port number in two ways: directly with an 8-bit constant, or indirectly with the DX register. For example, in the instruction

```
IN AL,61H   ; port number address is 61H
```

the port number is specified directly, but in

```
IN  AL,DX   ; port number is in DX register
```

it is specified indirectly. With the direct approach, the port number is limited to 8 bits; with the indirect approach it is limited to 16 bits. Figure 15.31 summarizes the I/O instructions on the Pentium.